

MySQL Tutorial

1. Database

2. Schema

schema is a logical structure that organizes and groups database objects, such as tables, views, indexes, stored procedures, functions, and more. It serves as a namespace within a database, allowing for better management, organization, and control of database objects.

1. Key Features of a Schema

1. Namespace:

- Objects within a schema are uniquely identified by their name and the schema they belong to.
- Example: `schema_name.table_name`.

2. Separation:

- Schemas allow the logical separation of objects within a database, making it easier to manage and secure them.

3. Ownership and Security:

- Each schema is owned by a specific database user or role.
- Permissions can be assigned at the schema level to control access to all objects within the schema.

4. Multi-Schema Support:

- Many databases allow multiple schemas in a single database, such as in **PostgreSQL**, **SQL Server**, **MySQL**, and **Oracle**.
-

2. Why Use Schemas?

1. Organization:

- Group related objects together, e.g., separating tables for different applications or modules.

2. **Security:**

- Grant permissions to specific schemas to limit access.

3. **Collaboration:**

- Multiple users can work within the same database while maintaining separate namespaces.

4. **Avoid Naming Conflicts:**

- Different schemas can have objects with the same name.

MySQL:

- In MySQL, a schema is synonymous with a database.
- Create a schema (or database):

Syntax:

CREATE DATABASE inventory;

- Use a schema:

USE inventory;

Schema vs Database

- **Schema:** A logical grouping of objects within a single database.
- **Database:** A container that holds multiple schemas and manages data storage.

3. Table:

Collection of rows and columns, Data's are presented in the rows and columns.

4. Few Databases:

1. Oracle
2. MS SQL
3. MySQL – invented by Oracle
4. DB2
5. Sybase

6. MongoDB
7. PostgreSQL

5. Why Database?

Before 1970, they stored in flat file.

In Database, almost single line which fetch the data you need

1. Fast
2. Reliable
3. Secure
4. Better memory conceptions

6. Stored procedure

A **stored procedure** is a prepared SQL code that **you can** save, so the code **can** be reused over and over again. ... **You can** also pass parameters to a **stored procedure**, so that the **stored procedure can** act based on the parameter value(s) that is passed

7. BLOB and CLOB

BLOB	CLOB
The full form of Blob is a B inary L arge O bject.	The full form of Clob is C haracter L arge O bject.
This is used to store large binary data.	This is used to store large textual data.
This stores values in the form of binary streams.	This stores values in the form of character streams.
Using this you can stores files like videos, images, gifs, and audio files.	Using this you can store files like text files, PDF documents, word documents etc.

BLOB	CLOB
<p>MySQL supports this with the following datatypes:</p> <ul style="list-style-type: none"> • TINYBLOB • BLOB • MEDIUMBLOB • LONGBLOB 	<p>MySQL supports this with the following datatypes:</p> <ul style="list-style-type: none"> • TINYTEXT • TEXT • MEDIUMTEXT • LONGTEXT
In JDBC API it is represented by java.sql.Blob Interface.	In JDBC it is represented by java.sql.Clob Interface.
The Blob object in JDBC points to the location of BLOB instead of holding its binary data.	The Blob object in JDBC points to the location of BLOB instead of holding its character data.
<p>To store Blob JDBC (PreparedStatement) provides methods like:</p> <ul style="list-style-type: none"> • setBlob() • setBinaryStream() 	<p>To store Clob JDBC (PreparedStatement) provides methods like:</p> <ul style="list-style-type: none"> • setClob() • setCharacterStream()
<p>And to retrieve (ResultSet) Blob it provides methods like:</p> <ul style="list-style-type: none"> • getBlob() • getBinaryStream 	<p>And to retrieve (ResultSet) Clob it provides methods like:</p> <ul style="list-style-type: none"> • getClob() • getCharacterStream()

8. Database Design

1.ER Models: Entities, Relationships, Attributes, and Cardinality

An Entity-Relationship (ER) model is a visual representation of data within a database. It uses diagrams to illustrate entities (objects or concepts), their attributes (properties), and the relationships between them. ER models are crucial for database

design, providing a clear blueprint for organizing and structuring data.

1. Key Components of ER Models

1. Entities

- **Definition:** Objects or "things" in the real world that can be distinctly identified.
- **Types:**
 - **Strong Entity:** An entity that exists independently of other entities (e.g., Student, Course).
 - **Weak Entity:** An entity that depends on a strong entity for its existence (e.g., Dependent linked to Employee).
- **Representation:** Rectangles in an ER diagram.

2. Attributes

- **Definition:** Properties or characteristics of an entity or relationship.
- **Types:**
 - **Simple Attribute:** Cannot be divided further (e.g., Name, Age).
 - **Composite Attribute:** Can be divided into sub-parts (e.g., FullName into FirstName, LastName).
 - **Derived Attribute:** Computed from other attributes (e.g., Age from DateOfBirth).
 - **Multi-Valued Attribute:** Can have multiple values for a single entity (e.g., PhoneNumbers).
- **Key Attribute (Primary Key):** Uniquely identifies each instance of an entity. Underlined in the ER diagram.
- **Representation:** Ovals connected to entities or relationships.

3. Relationships

- **Definition:** Associations between two or more entities.
- **Types:**
 - **Unary Relationship:** Involves only one entity type (e.g., Employee manages Employee).
 - **Binary Relationship:** Involves two entity types (e.g., Student enrolls in Course).
 - **Ternary Relationship:** Involves three entity types (e.g., Doctor treats Patient at a Hospital).
- **Representation:** Diamonds in an ER diagram.

4. Cardinality

Definition:

- Specifies the number of instances of one entity that can or must be associated with an instance of another entity.






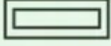
Types:

- **One-to-One (1:1):** One entity instance is related to one instance of another entity.
 - Example: Person and Passport.
- **One-to-Many (1:N):** One entity instance is related to multiple instances of another entity.
 - Example: Teacher teaches multiple Students.
- **Many-to-One (N:1):** The reverse of one-to-many.
- **Many-to-Many (M:N):** Multiple instances of one entity are related to multiple instances of another entity.
 - Example: Students enroll in Courses.

Cardinality is often represented using Crow's Foot notation:

- | represents "one".
 - O represents "zero".
 - > represents "many".
-

2. Diagram Representation

Figures	Symbols	Represents
Rectangle		Entities in ER Model
Ellipse		Attributes in ER Model
Diamond		Relationships among Entities
Line		Attributes to Entities and Entity Sets with Other Relationship Types
Double Ellipse		Multi-Valued Attributes
Double Rectangle		Weak Entity

Component	Symbol	Example
Entity	Rectangle	Student, Employee
Weak Entity	Double Rectangle	Dependent
Attribute	Oval	Name, Age
Derived Attribute	Dashed Oval	Age (from DOB)
Multi-Valued Attr.	Double Oval	PhoneNumbers
Relationship	Diamond	Enrolled
Cardinality	1:1, 1:N, M:N (next to relationship lines)	Teacher teaches Students

5. Examples

1. University Database

- **Entities:** Student, Course, Professor.
- **Attributes:**
 - Student: StudentID, Name, Major.
 - Course: CourseID, Title, Credits.
 - Professor: ProfessorID, Name, Department.
- **Relationships:**

- Student enrolls in Course.
- Professor teaches Course.
- **Cardinality:**
 - Student can enroll in many Courses (1:N).
 - Course can be taught by one Professor (N:1).

2. Employee-Dependent Database

- **Entities:** Employee, Dependent.
 - **Attributes:**
 - Employee: EmployeeID, Name, Position.
 - Dependent: Name, Relationship.
 - **Relationships:**
 - Employee has Dependent (1:N).
 - **Cardinality:**
 - One Employee can have multiple Dependents.
-

3. Steps to Create an ER Model

1. **Identify Entities:** Define the main objects in the system.
 2. **Identify Relationships:** Determine how entities are related.
 3. **Identify Attributes:** List properties for each entity and relationship.
 4. **Determine Cardinality:** Specify the number of associations between entities.
 5. **Draw the Diagram:** Represent entities, relationships, and attributes graphically.
-

4. Key Considerations

- **Normalization:** Helps ensure the model avoids redundancy and maintains consistency.
- **Primary Keys:** Unique identifiers for each entity.
- **Foreign Keys:** Represent relationships between entities.

5. Importance of ER Models:

- **Clear Communication:** Facilitates communication between database designers, developers, and users.
- **Database Design:** Provides a solid foundation for designing relational databases.
- **Data Consistency:** Helps identify data redundancies and inconsistencies.
- **Documentation:** Serves as valuable documentation for the database structure.

2. Normal Forms and Normalization Process

Normalization is a process in database design that organizes data to reduce redundancy and improve data integrity. It involves decomposing a database into smaller, related tables and ensuring data dependencies are logical.

The process is guided by rules called **Normal Forms (NF)**, which range from the least strict (**1NF**) to the most strict (**5NF** and beyond).

1. Objectives of Normalization

1. **Eliminate redundancy:** Minimize duplicate data.
 2. **Ensure data integrity:** Protect against logical inconsistencies.
 3. **Simplify queries:** Enhance efficiency in data manipulation.
 4. **Improve scalability:** Make the database easier to expand and maintain.
-

2. The Normal Forms

1. First Normal Form (1NF):

- **Definition:** A table is in 1NF if:
 1. Each cell contains a single value (atomic values).
 2. Each row is unique (has a primary key).

- **Example:**

1. **Not in 1NF:**

StudentID	Name	Subjects
1	Alice	Math, Science

2. **In 1NF:**

StudentID	Name	Subject
1	Alice	Math
1	Alice	Science

3. Second Normal Form (2NF):

- **Definition:** A table is in 2NF if:

1. It is in 1NF.
2. All non-key attributes are fully dependent on the primary key.

A **non-key attribute** is an attribute (or column) in a table that is **not part of any candidate key**.

A **candidate key** is a set of **one or more columns that can uniquely identify each row** in a table. If a column is part of a candidate key, it is referred to as a **key attribute**. Any other attribute that is not part of a candidate key is considered a **non-key attribute**.

3. Example:

Consider the following table Employees:

EmployeeID	Name	Department	Salary
1	Alice	HR	50000

EmployeeID	Name	Department	Salary
2	Bob	IT	60000
3	Charlie	Marketing	55000

1. **Candidate Key:**

- EmployeeID (because it uniquely identifies each row).

2. **Key Attribute:**

- EmployeeID (as it is part of the candidate key).

3. **Non-Key Attributes:**

- Name, Department, Salary (because they are not part of the candidate key and do not uniquely identify rows).

• **Example:**

1. **Not in 2NF** (partial dependency):

StudentID	CourseID	CourseName
1	C001	Math

- CourseName depends only on CourseID.

2. **In 2NF:**

- Split the table:

1. StudentCourses:

StudentID	CourseID
1	C001

2. Courses:

CourseID	CourseName
C001	Math

3. Third Normal Form (3NF):

- **Definition:** A table is in 3NF if:
 1. It is in 2NF.
 2. All attributes are dependent only on the primary key (no transitive dependency).
- **Example:**
 1. **Not in 3NF** (transitive dependency):

StudentID	AdvisorID	AdvisorName
1	A001	Dr. Smith

- AdvisorName depends on AdvisorID, which depends on StudentID.

2. In 3NF:

- Split the table:
 1. Students:

StudentID	AdvisorID
1	A001

2. Advisors:

AdvisorID	AdvisorName
A001	Dr. Smith

4. Boyce-Codd Normal Form (BCNF):

- **Definition:** A table is in BCNF if:
 1. It is in 3NF.
 2. Every determinant is a candidate key (handles situations where 3NF fails).
- **Example:**

1. Not in BCNF:

TeacherID	Subject	Department
1	Math	Science

- Subject determines Department.

2. In BCNF:

- Split the table:

1. TeacherSubjects:

TeacherID	Subject
1	Math

2. Subjects:

Subject	Department
Math	Science

5. Fourth Normal Form (4NF):

- **Definition:** A table is in 4NF if:
 1. It is in BCNF.
 2. It has **no multi-valued dependencies (one attribute determines a set of other attributes independently)**.
-

6. Fifth Normal Form (5NF):

- **Definition:** A table is in 5NF if:
 1. It is in 4NF.
 2. It **cannot be further decomposed without losing data**.
-

4. Normalization Process

1. Unnormalized Data:

- Start with a table that may have repeating groups or arrays.
 - 2. **1NF**:
 - Remove repeating groups, ensure atomic values.
 - 3. **2NF**:
 - Remove partial dependencies.
 - 4. **3NF**:
 - Remove transitive dependencies.
 - 5. **BCNF, 4NF, 5NF** (optional):
 - Apply higher normal forms as needed for advanced cases.
-

5. Advantages of Normalization

- Reduces redundancy.
- Minimizes update, delete, and insert anomalies.
- Improves data integrity and consistency.

6. Disadvantages of Normalization

- Increased complexity.
- Requires more joins, potentially impacting performance for large queries.
- May lead to more tables, making management harder.

3. Logical and Physical Database Design

4. Logical and Physical Database Design

Database design is a critical process that ensures data is structured efficiently and can meet business requirements. It involves two main stages: **logical design** and **physical design**.

9. 1. Logical Database Design

5. 1.1 Definition

Logical design is the process of creating a conceptual model of the database without focusing on its physical implementation. It defines the structure of the data, relationships, and constraints to meet business requirements.

6. 1.2 Key Components

4. 1.2.1 Entity-Relationship Diagram (ERD)

- A visual representation of entities (tables), attributes (columns), and relationships.

Example:

- Entities: Customer, Order
- Attributes: Customer_ID, Name, Order_ID, Date
- Relationships: A Customer places one or more Orders.

5. 1.2.2 Normalization

- A technique to eliminate data redundancy and ensure data integrity.
- Normal Forms (NF):
 - **1NF**: Ensure atomic values and unique rows.
 - **2NF**: Remove partial dependencies.
 - **3NF**: Remove transitive dependencies.

Example: Splitting a single table:

Unnormalized Table:

Order_ID	Customer_Name	Customer_Address
----------	---------------	------------------

1NF:

Order_ID	Customer_ID	Customer_Name	Customer_Address
1	101	John Doe	123 Street

2NF:

Customer_ID	Customer_Name	Customer_Address
-------------	---------------	------------------

| 101 | John Doe | 123 Street |

| Order_ID | Customer_ID |

6. 1.2.3 Business Rules

- Define constraints like unique keys, primary keys, and foreign keys.

7. 1.2.4 Data Types

- Assign appropriate data types (e.g., VARCHAR, INT, DATE).
-

7. 1.3 Advantages of Logical Design

1. **Improves Data Integrity:** Ensures consistent data through constraints.
 2. **Facilitates Communication:** Bridges the gap between business requirements and technical implementation.
 3. **Scalable Structure:** Supports future growth without redundancy.
-

10. 2. Physical Database Design

8. 2.1 Definition

Physical design translates the logical design into a specific database schema tailored to the hardware and database management system (DBMS). It optimizes storage and performance.

9. 2.2 Key Components

8. 2.2.1 Physical Schema

- Specifies how data is stored, including tables, indexes, and partitions.

9. 2.2.2 Indexing

- Improves query performance by creating indexes on frequently queried columns.
 - **Clustered Index:** Determines the physical order of rows in a table.

- **Non-Clustered Index:** Provides a logical order for data without altering physical storage.

10. 2.2.3 Partitioning

- Divides a large table into smaller, manageable pieces.
 - **Horizontal Partitioning:** Splits rows.
 - **Vertical Partitioning:** Splits columns.

11. 2.2.4 Storage and Compression

- Choose storage formats and enable compression to save space and enhance performance.

12. 2.2.5 Query Optimization

- Analyze execution plans and refine queries for efficiency.
-

10. 2.3 Implementation Factors

13. 2.3.1 DBMS-Specific Features

- Leverage specific features of the chosen DBMS (e.g., MySQL InnoDB, PostgreSQL tablespaces).

14. 2.3.2 Hardware Configuration

- Consider disk type (SSD vs. HDD), memory, and CPU when designing for performance.

15. 2.3.3 Backup and Recovery

- Plan for data backup and recovery mechanisms to ensure reliability.
-

11. 2.4 Advantages of Physical Design

1. **Optimized Performance:** Tailored storage and indexing reduce query times.
 2. **Efficient Storage:** Saves disk space with compression and partitioning.
 3. **Scalable Deployment:** Accommodates large-scale data efficiently.
-

11. 3. Key Differences Between Logical and Physical Design

Aspect	Logical Design	Physical Design
Focus	Data structure and relationships	Storage, performance, and DBMS specifics
Abstraction Level	High	Low
Tools Used	ER diagrams, normalization	Indexing, partitions, execution plans
DBMS Dependence	Independent	Dependent
Purpose	Ensures data integrity and consistency	Ensures performance and scalability

12. 4. Example

12. Logical Design

- ERD:
 - Entities: Employee, Department
 - Attributes: Employee_ID, Name, Department_ID, Department_Name
 - Relationship: One department has many employees.

13. Physical Design

- **Tables:**
 - Employee (indexed on Employee_ID)
 - Department (indexed on Department_ID)
- **Indexes:**
 - CREATE INDEX idx_employee_name ON Employee (Name);
- **Partitioning:**

- CREATE TABLE Employee PARTITION BY RANGE (HireDate) (
 - PARTITION p0 VALUES LESS THAN ('2000-01-01'),
 - PARTITION p1 VALUES LESS THAN ('2010-01-01'),
 - PARTITION p2 VALUES LESS THAN MAXVALUE
 -);
-

13. 5. Best Practices

14. Logical Design Best Practices

1. **Focus on Business Needs:** Align design with business processes.
2. **Normalize Judiciously:** Avoid over-normalization, which may impact performance.
3. **Document Relationships:** Clearly define constraints and dependencies.

15. Physical Design Best Practices

1. **Optimize Index Usage:** Avoid excessive indexes that may slow writes.
 2. **Monitor Query Performance:** Use tools like EXPLAIN to analyze execution plans.
 3. **Adapt to Workload:** Partition data and fine-tune storage based on access patterns.
-

By understanding and carefully implementing logical and physical database designs, you can create efficient, scalable, and secure databases that meet organizational needs. Let me know if you'd like to dive deeper into any specific area!

14. SQL Syntax and Statement Structure

1. SQL Keywords and Identifiers

In SQL, **keywords** and **identifiers** play a crucial role in writing and understanding queries. Here's a breakdown of these terms:

1. Keywords

- **Definition:** Reserved words in SQL that have a specific meaning and functionality.
- **Purpose:** Used to perform various operations like defining, manipulating, and querying data.
- **Examples:**
 - **Data Definition Language (DDL):**
 - CREATE, DROP, , TABLE
 - **Data Manipulation Language (DML):**
 - SELECT, INSERT, UPDATE, DELETE
 - **Data Query Language (DQL):**
 - SELECT, WHERE, ORDER BY, GROUP BY
 - **Data Control Language (DCL):**
 - GRANT, REVOKE
 - **Transaction Control Language (TCL):**
 - COMMIT, ROLLBACK, SAVEPOINT

Common SQL Keywords

Category	Keywords
DDL	CREATE, DROP, ALTER, TRUNCATE
DML	INSERT, UPDATE, DELETE, MERGE
DQL	SELECT, DISTINCT, WHERE, HAVING
DCL	GRANT, REVOKE
TCL	COMMIT, ROLLBACK, SAVEPOINT
Miscellaneous	CASE, WHEN, THEN, AS, NULL, LIKE, IN, BETWEEN

2. Identifiers

- **Definition:** Names used to identify database objects such as tables, columns, databases, schemas, indexes, views, and constraints.

- **Purpose:** Enable users to reference database objects in queries.
- **Examples:**
 - Table names: employees, sales_data
 - Column names: employee_id, salary
 - Schema names: hr, finance

Rules for Identifiers

1. Naming Conventions:

- Should begin with a letter (a-z, A-Z).
- Can include letters, numbers (0-9), and underscores (_).
- Avoid starting with numbers or using special characters like \$, @, #.

2. Case Sensitivity:

- SQL keywords **are case-insensitive** (SELECT is the same as select).
- Identifiers' case sensitivity depends on the database (e.g., MySQL is case-insensitive by default, PostgreSQL is case-sensitive).

3. Reserved Words:

- Avoid using SQL keywords as identifiers unless enclosed in delimiters (e.g., backticks in MySQL or double quotes in PostgreSQL).

SELECT "SELECT" FROM my_table; -- Using a keyword as a column name

4. Length Restrictions:

- Varies by database, but typically identifiers should not exceed 128 characters.

◦

Differences Between Keywords and Identifiers

Aspect	Keywords	Identifiers
Purpose	Defines SQL	Names database

Aspect	Keywords	Identifiers
	operations	objects
Reserved	Reserved and predefined	User-defined
Examples	SELECT, WHERE, CREATE	employee_id, sales_data
Case Sensitivity	Case-insensitive (mostly)	Depends on the database

2. Practical Examples

Keywords in Action

```
SELECT employee_id, first_name, salary
FROM employees
WHERE salary > 50000
ORDER BY salary DESC;
```

Keywords: SELECT, FROM, WHERE, ORDER BY, DESC.

Identifiers in Action

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    salary DECIMAL(10, 2)
);
```

Identifiers: employees, employee_id, first_name, salary.

2.Statement, Clauses and Predicates

In SQL, **statements**, **clauses**, and **predicates** are fundamental components of writing queries. Here's what each of these terms means:

1. Statement

An **SQL statement** is a **complete instruction that performs a specific task**. It can be a query, data modification, or a command to manage the database structure.

Examples:

- `SELECT * FROM employees;` (query statement)
 - `INSERT INTO employees (id, name) VALUES (1, 'John');` (data modification statement)
 - `CREATE TABLE employees (id INT, name VARCHAR(50));` (data definition statement)
-

2. Clause

A **clause** is a component of an SQL statement. It defines specific parts of the SQL statement and gives additional instructions or conditions.

Common SQL Clauses:

- **SELECT:** Specifies the columns to retrieve.
Example: `SELECT name, salary FROM employees;`
 - **FROM:** Specifies the source table.
Example: `FROM employees`
 - **WHERE:** Filters rows based on a condition.
Example: `WHERE salary > 5000`
 - **GROUP BY:** Groups rows sharing a value into summary rows.
Example: `GROUP BY department`
 - **HAVING:** Filters groups after aggregation.
Example: `HAVING COUNT(*) > 5`
 - **ORDER BY:** Sorts the result.
Example: `ORDER BY name ASC`
-

3. Predicate

A **predicate** is a **condition or logical expression used within clauses** like WHERE, HAVING, or ON (in joins). It evaluates to TRUE, FALSE, or NULL.

Examples of Predicates:

- **Comparison Predicate:**
Example: salary > 5000
- **Range Predicate:**
Example: age BETWEEN 30 AND 50
- **Membership Predicate:**
Example: department IN ('IT', 'HR')
- **Null Predicate:**
Example: manager_id IS NULL
- **Pattern Matching Predicate:**
Example: name LIKE 'J%'
- **Existence Predicate:**
Example: EXISTS (SELECT 1 FROM departments WHERE manager_id = employees.id)

3. Example SQL Query with Clauses and Predicates:

```
SELECT name, department, salary      -- SELECT Clause
FROM employees                      -- FROM Clause
WHERE salary > 5000 AND department = 'IT' -- WHERE Clause with Predicates
GROUP BY department                -- GROUP BY Clause
HAVING AVG(salary) > 6000           -- HAVING Clause with Predicate
ORDER BY salary DESC;               -- ORDER BY Clause
```

- **Clauses:** SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY
- **Predicates:** salary > 5000, department = 'IT', AVG(salary) > 6000

3. Comments and Formatting Best Practices

Good comments and proper formatting in SQL enhance readability, maintainability, and collaboration. Here are some **best practices for commenting and formatting SQL code**:

1. Comments in SQL

SQL supports two types of comments:

Single-line Comments

Begin with -- and continue until the end of the line.

Example:

```
-- This query retrieves all employees with a salary greater than 5000
SELECT name, salary
FROM employees
WHERE salary > 5000;
```

Multi-line Comments

Enclosed between /* and */. Use for longer explanations or block comments.

Example:

```
/*
This query retrieves employees who are:
1. Working in the IT department
2. Earning a salary greater than 5000
*/
SELECT name, salary
FROM employees
WHERE salary > 5000 AND department = 'IT';
```

Best Practices for Comments

- **Be concise but meaningful:** Explain the "why," not just the "what."
 - -- This is a query.
 - -- Retrieve IT employees earning above 5000.
 - **Avoid redundant comments:** Don't state the obvious.
 - -- Select name and salary from employees.
 - -- Fetch employee details for salary analysis.
 - **Update comments when code changes:** Outdated comments can be misleading.
-

2. SQL Formatting Best Practices

a. Use Consistent Indentation

Proper indentation improves readability, especially for complex queries.

Example:

```
SELECT name, salary
FROM employees
WHERE department = 'IT'
      AND salary > 5000
ORDER BY salary DESC;
```

b. Capitalize Keywords

Write SQL keywords in uppercase for clarity.

Example:

```
SELECT name, department
FROM employees
WHERE salary > 5000;
```

c. Use Line Breaks for Logical Segments

Break long statements into multiple lines for better readability.

Example:

```
SELECT name, department, salary
FROM employees
WHERE department = 'HR'
      AND salary > 5000
ORDER BY salary DESC;
```

d. Align Conditions for Readability

Align AND/OR conditions under the WHERE clause for better clarity.

Example:

```
SELECT name, department, salary
FROM employees
WHERE department = 'HR'
      AND salary > 5000
      AND hire_date > '2020-01-01';
```

e. Use Aliases Wisely

Use short, meaningful aliases to simplify column and table references.

Example:

```
SELECT e.name, e.salary, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id;
```

f. Avoid Hardcoding Values

Use variables or placeholders for better maintainability.

Example:

```
SELECT name, salary
FROM employees
WHERE salary > @min_salary;  -- Use parameterized queries
```

g. Group Clauses Logically

Write clauses in the typical SQL order:

1. **SELECT**
2. **FROM**
3. **JOIN** (if any)
4. **WHERE**
5. **GROUP BY**
6. **HAVING**
7. **ORDER BY**

Example:

```
SELECT name, department, COUNT(*) AS total_employees
FROM employees
WHERE department IN ('IT', 'HR')
GROUP BY department
HAVING COUNT(*) > 5
ORDER BY total_employees DESC;
```

3. Example: Well-Formatted and Commented SQL

```

-- Retrieve employees in IT and HR departments
-- who earn more than 5000 and joined after 2020
SELECT
    e.name AS employee_name,      -- Employee name
    e.salary,                     -- Employee salary
    d.department_name             -- Department name
FROM
    employees e
JOIN
    departments d ON e.department_id = d.department_id
WHERE
    e.salary > 5000                -- Filter for salaries above 5000
    AND d.department_name IN ('IT', 'HR') -- Departments: IT, HR
    AND e.hire_date > '2020-01-01' -- Hired after 2020
ORDER BY
    e.salary DESC;                -- Sort by salary in descending order

```

By following these practices, your SQL code becomes more structured, understandable, and maintainable for you and your team.

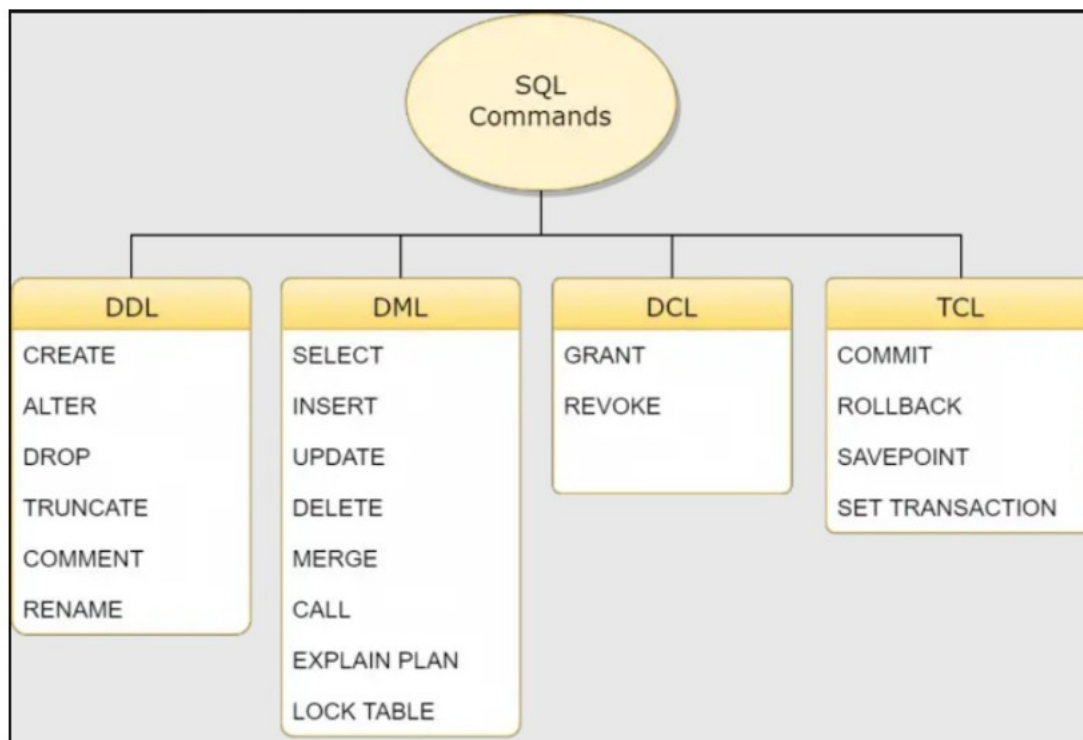
15. Data Types:

Data type	Description
CHAR(size)	A FIXED length string (can contain letters, numbers, and special characters). The size parameter specifies the column length in characters - can be from 0 to 255. Default is 1
VARCHAR(size)	A VARIABLE length string (can contain letters, numbers, and special characters). The <i>size</i> parameter specifies the maximum column length in characters - can be from 0 to 65535
BINARY(size)	Equal to CHAR(), but stores binary byte strings. The size parameter specifies the column length in bytes. Default is 1
BOOL	Zero is considered as false, nonzero values are

	considered as true.
BOOLEAN	Equal to BOOL
SMALLINT(size)	A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The size parameter specifies the maximum display width (which is 255)
MEDIUMINT(size)	A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The size parameter specifies the maximum display width (which is 255)
INT(size)	A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The size parameter specifies the maximum display width (which is 255)
INTEGER(size)	Equal to INT(size)
BIGINT(size)	A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The size parameter specifies the maximum display width (which is 255)
FLOAT(size, d)	A floating point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter. This syntax is deprecated in MySQL 8.0.17 , and it will be removed in future MySQL versions
FLOAT(p)	A floating point number. MySQL uses the p value to determine whether to use FLOAT or DOUBLE for the resulting data type. If p is from 0 to 24, the data type becomes FLOAT(). If p is from 25 to 53, the data type becomes DOUBLE()
DOUBLE(size, d)	A normal-size floating point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in

	the d parameter
DOUBLE PRECISION(size , d)	
DECIMAL(size, d)	An exact fixed-point number. The total number of digits is specified in size. The number of digits after the decimal point is specified in the d parameter. The maximum number for size is 65. The maximum number for d is 30. The default value for size is 10. The default value for d is 0.
DEC(size, d)	Equal to DECIMAL(size,d)

16. SQL: Different Type of Queries



1.DDL - Data Definition Language

1. Various SQL Commands

1. Create
2. Alter
3. Drop
4. Truncate

5. Rename

2. CREATE INDEX and DROP INDEX

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. They essentially create a shortcut for finding specific rows in a table without having to scan the entire table. CREATE INDEX and DROP INDEX are SQL commands used to manage indexes on database tables.

1. CREATE INDEX

The CREATE INDEX statement is used to create an index on a table.

Syntax:

```
CREATE INDEX index_name ON table_name(column_name);
```

Example:

```
CREATE INDEX idx_student_name ON students(name);
```

- **idx_student_name:** The name of the index.
- **students:** The table name.
- **name:** The column to be indexed.

Types of Indexes

1. **Unique Index:** Ensures all values in the column are unique.

```
CREATE UNIQUE INDEX idx_unique_email ON students(email);
```

2. **Composite Index:** Creates an index on multiple columns.

```
CREATE INDEX idx_composite ON students(last_name, first_name);
```

3. **Full-Text Index:** Used for full-text search.

```
CREATE FULLTEXT INDEX idx_fulltext ON students(bio);
```

4. **Spatial Index** (for GIS data types):


```
CREATE SPATIAL INDEX idx_location ON locations(geometry_column);
```

2. DROP INDEX

The DROP INDEX statement is used to remove an index from a table.

Syntax:

```
DROP INDEX index_name ON table_name;
```

Example:

```
DROP INDEX idx_student_name ON students;
```

This removes the idx_student_name index from the students table.

2. Notes

- Indexes speed up SELECT queries but may slow down INSERT, UPDATE, and DELETE operations due to the overhead of maintaining the index.
 - Use indexes wisely for frequently queried columns or those used in WHERE, ORDER BY, or GROUP BY clauses.
 - You cannot directly update an index; instead, you need to drop and recreate it.
-

3. View Existing Indexes

To view the indexes of a table:

Syntax:

```
SHOW INDEX FROM table_name;
```

Example:

```
SHOW INDEX FROM students;
```

4. Best Practices

1. Index only when necessary (e.g., on primary keys and foreign keys).
2. Avoid over-indexing as it increases storage and maintenance costs.
3. Use composite indexes for queries involving multiple columns.

When to Use Indexes:

- Columns frequently used in WHERE clauses (especially with =, >, <, >=, <=, BETWEEN, IN, LIKE).
- Columns used in JOIN conditions.
- Columns used in ORDER BY clauses.

When NOT to Use Indexes:

- Small tables. The overhead of maintaining the index might outweigh the benefits.
- Columns with high cardinality (many distinct values) where you frequently retrieve a large portion of rows. A full table scan might be more efficient.
- Columns that are frequently updated. Every update to the indexed column requires an update to the index as well, which can impact performance.
- Columns used in WHERE clauses with functions applied to them. The database might not be able to use the index effectively.

Key Considerations:

- Indexes improve read performance but can slightly slow down write operations (inserts, updates, deletes) because the index needs to be updated as well.
- Indexes consume storage space.
- Over-indexing can negatively impact performance. Choose indexes carefully based on your query patterns.

3. Data Types and Table Storage Engines

Refer [Section 9](#) for Data Types

Table Storage Engines

5. DML - Data Manipulation Language

1. Various DML SQL Commands

1. Insert
2. Update
3. Delete
4. Merge

2. BULK INSERT

Here's a detailed overview of **BULK INSERT** and data import/export techniques in MySQL, focusing on efficient methods to handle large datasets.

1. BULK INSERT in MySQL

While MySQL does not have a BULK INSERT command like SQL Server, it achieves similar functionality using **LOAD DATA INFILE** and **mysqlimport**.

6. 2. Using **LOAD DATA INFILE** for Bulk Inserts

The LOAD DATA INFILE statement imports data from a text file (e.g., CSV) into a database table. It is the fastest way to load large datasets in MySQL.

3. Syntax:

```
LOAD DATA INFILE 'file_path'
INTO TABLE table_name
FIELDS TERMINATED BY ',' -- Field separator
OPTIONALLY ENCLOSED BY '"' -- Enclosing character (e.g., for strings)
LINES TERMINATED BY '\n' -- Row separator
IGNORE n ROWS; -- Skip header or initial rows
```

Example:

Import data from employees.csv into a table named employees:

```
LOAD DATA INFILE '/path/to/employees.csv'
INTO TABLE employees
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS; -- Skip the header row
```

4. Data Import Technique

3. Using mysqlimport Utility

The mysqlimport utility is a command-line tool for importing files into MySQL tables. It is a wrapper for the LOAD DATA INFILE command.

Syntax:

```
mysqlimport --host=hostname --user=username --password=yourpassword \
--fields-terminated-by=',' --lines-terminated-by='\n' database_name /path/to/file.csv
```

Example:

Import employees.csv into the employees table:

```
mysqlimport --fields-terminated-by=',' --lines-terminated-by='\n' \
--ignore-lines=1 my_database /path/to/employees.csv
```

5. Data Export Technique

Export Data Using SELECT INTO OUTFILE

The SELECT INTO OUTFILE statement exports query results to a file on the MySQL server.

Syntax:

```
SELECT columns
INTO OUTFILE 'file_path'
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n'
FROM table_name
WHERE conditions;
```

Example:

Export the employees table to a CSV file:

```
SELECT id, name, salary, department
INTO OUTFILE '/path/to/employees.csv'
FIELDS TERMINATED BY ','
OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\n'
FROM employees;
```

5. Data Import/Export with Tools

a. MySQL Workbench

1. Data Import:

- Use the **Data Import Wizard** to load CSV files into tables.
- Navigate to **Server > Data Import**.

2. Data Export:

- Use the **Data Export Wizard** to export tables to CSV or SQL files.
- Navigate to **Server > Data Export**.

b. MySQL Dump Utility

- Use mysqldump to export databases or tables.

Export a Database:

```
mysqldump -u username -p database_name > database_backup.sql
```

Export a Table:

```
mysqldump -u username -p database_name table_name > table_backup.sql
```

6. Error Handling and Permissions

6. Common Errors and Solutions:

- **ERROR 1290 (secure_file_priv restriction):**

- MySQL restricts file imports/exports to a specific directory. Check secure_file_priv:

```
SHOW VARIABLES LIKE 'secure_file_priv';
```

- Move your file to the permitted directory or disable the restriction.

- **Enable LOAD DATA LOCAL INFILE:**
 - For local files, enable the --local-infile option:

```
mysql --local-infile=1 -u username -p
```

7. Granting Permissions:

Ensure the MySQL user has the necessary permissions:

```
GRANT FILE ON *.* TO 'username'@'host';
FLUSH PRIVILEGES;
```

7. Optimizing Bulk Data Import

1. Disable Indexes Temporarily:

```
ALTER TABLE employees DISABLE KEYS;
LOAD DATA INFILE '/path/to/employees.csv'
INTO TABLE employees
FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n';
ALTER TABLE employees ENABLE KEYS;
```

2. **Batch Processing:** If the file is too large, split it into smaller chunks.
3. **Use Transactions:** Wrap LOAD DATA operations in a transaction for better control.
4. **Use MyISAM for Temporary Loads:** If possible, use a temporary MyISAM table for loading and later transfer data to the target InnoDB table.

8. Practical Workflow

8. a. Import Workflow:

1. Prepare the file (employees.csv):

```
id,name,salary,department
1,John,50000,IT
2,Jane,60000,HR
```

2. Create a table:

```
CREATE TABLE employees (  
    id INT,  
    name VARCHAR(50),  
    salary DECIMAL(10, 2),  
    department VARCHAR(50)  
);
```

3. Import the file:

```
LOAD DATA INFILE '/path/to/employees.csv'  
INTO TABLE employees  
FIELDS TERMINATED BY ','  
OPTIONALLY ENCLOSED BY ''''  
LINES TERMINATED BY '\n'  
IGNORE 1 ROWS;
```

b. Export Workflow:

1. Export the table to a file:

```
SELECT *  
INTO OUTFILE '/path/to/employees.csv'  
FIELDS TERMINATED BY ','  
OPTIONALLY ENCLOSED BY ''''  
LINES TERMINATED BY '\n'  
FROM employees;
```

3.DQL - Data Query Language

1. Select

4. TCL - Transaction Control Language

1. Transaction
2. Commit – save changes permanently
3. Rollback – undo changes
4. Savepoint

1.Transaction Management

Transaction management is a crucial aspect of database systems and other applications that involve data manipulation. It ensures that **data remains consistent and reliable even when**

multiple operations are performed concurrently or in the face of system failures.

What is a Transaction?

In the context of databases, a transaction is a sequence of one or more operations performed as a single logical unit of work. These operations could include:

- **Reading data:** Retrieving information from the database.
- **Writing data:** Inserting, updating, or deleting data in the database.

ACID Properties

To guarantee data integrity, transactions must adhere to the ACID properties:

- **Atomicity:** A transaction is treated as a single, indivisible unit of work. Either all operations within the transaction are completed successfully, or none are. If any part of the transaction fails, the entire transaction is rolled back, and the database is restored to its previous state.
- **Consistency:** A transaction must maintain the database's integrity constraints. It ensures that the database transitions from one valid state to another. If a transaction violates any constraints, it is rolled back.
- **Isolation:** Transactions should be isolated from each other, meaning that concurrent transactions should not interfere with each other's execution. Each transaction should operate as if it were the only transaction running on the database.
- **Durability:** Once a transaction is committed (successfully completed), the changes made to the database are permanent and will survive even system failures such as power outages or crashes.

Why is Transaction Management Important?

Transaction management is essential for several reasons:

- **Concurrency Control:** In environments where multiple users or applications access the database simultaneously, transaction management prevents data corruption and ensures that transactions are executed in a consistent and predictable manner.
- **Error Recovery:** If a system failure occurs during a transaction, transaction management allows the database to recover to a consistent state by rolling back any incomplete transactions.

- **Data Integrity:** By enforcing the ACID properties, transaction management guarantees that data remains accurate and reliable, even in the face of errors or concurrent access.

Transaction Management Techniques

Database systems employ various techniques to manage transactions, including:

- **Concurrency Control Mechanisms:** These mechanisms, such as locking and timestamping, regulate the access of concurrent transactions to shared data, preventing conflicts and ensuring isolation.
- **Logging:** The database system maintains a log of all transaction operations. This log is used for recovery purposes in case of system failures.
- **Two-Phase Commit (2PC):** This protocol ensures that distributed transactions (transactions involving multiple databases) are committed atomically.

Example

Consider a banking transaction where money is transferred from account A to account B. This transaction involves two operations:

1. Debit the amount from account A.
2. Credit the amount to account B.

If a system failure occurs after the first operation but before the second, transaction management ensures that the debit operation is rolled back, preventing an inconsistency where money is deducted from one account but not added to the other.

1. Commands for Transaction Management:

- **START TRANSACTION:** Begins a new transaction.
- **COMMIT:** Saves changes made during the transaction.
- **ROLLBACK:** Undoes changes made during the transaction.
- **SAVEPOINT:** Sets a savepoint within a transaction.
- **ROLLBACK TO SAVEPOINT:** Rolls back to a specific savepoint.

5. Transaction Management Commands

1. START TRANSACTION

- Begins a new transaction.

Syntax:

```
START TRANSACTION;
```

2. COMMIT

- Permanently saves changes made during the transaction.

Syntax:

```
COMMIT;
```

3. ROLLBACK

- Reverts all changes made during the transaction since the last commit.

Syntax:

```
ROLLBACK;
```

4. SAVEPOINT

- Creates a checkpoint within a transaction that you can roll back to without rolling back the entire transaction.

Syntax:

```
SAVEPOINT savepoint_name;
```

5. ROLLBACK TO SAVEPOINT

- Reverts changes to the specified savepoint.

Syntax:

```
ROLLBACK TO SAVEPOINT savepoint_name;
```

6. RELEASE SAVEPOINT

- Removes a savepoint.

Syntax:

```
RELEASE SAVEPOINT savepoint_name;
```

7. SET AUTOCOMMIT

- By default, MySQL runs in autocommit mode, where each statement is treated as a transaction and committed automatically.

To disable autocommit:

```
SET AUTOCOMMIT = 0;
```

To enable autocommit:

```
SET AUTOCOMMIT = 1;
```

6. Example of Transaction Management

Basic Example

```
START TRANSACTION;  
  
UPDATE accounts SET balance = balance - 500 WHERE account_id = 1;  
UPDATE accounts SET balance = balance + 500 WHERE account_id = 2;  
  
COMMIT;
```

- **Explanation:** The transaction ensures both updates succeed. If any error occurs, changes are not committed.

Example with Rollback

```

START TRANSACTION;

UPDATE accounts SET balance = balance - 500 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 500 WHERE account_id = 2;

-- Simulate an error
IF (ROW_COUNT() = 0) THEN
    ROLLBACK;
ELSE
    COMMIT;
END IF;

```

- **Explanation:** If no rows are updated, the transaction is rolled back.

Using SAVEPOINT

```

START TRANSACTION;

UPDATE inventory SET stock = stock - 10 WHERE product_id = 101;
SAVEPOINT step1;

UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;

-- Something goes wrong
ROLLBACK TO SAVEPOINT step1;

COMMIT;

```

- **Explanation:** The rollback reverts to step1, but changes before the savepoint remain.

7. Benefits of Transactions

- Prevents partial updates.
- Ensures data integrity.
- Allows for error handling in complex operations.
- Provides greater control over database operations.

2. Implicit and Explicit Transactions

Transaction management in SQL can be categorized into two types: **implicit transactions** and **explicit transactions**. The distinction

lies in how and when the transaction boundaries (start and end) are defined.

8. Implicit Transactions

1. Definition:

An **implicit transaction** is automatically started by the database when certain SQL statements are executed. The transaction remains active until you explicitly commit or roll it back.

2. Characteristics:

- The transaction begins automatically when executing certain SQL commands.
- Requires manual COMMIT or ROLLBACK to end the transaction.
- Suitable for environments where transaction control is necessary but not explicitly defined.

3. Common SQL Statements That Trigger Implicit Transactions:

- INSERT
- UPDATE
- DELETE
- MERGE
- SELECT INTO

4. Enabling Implicit Transactions:

To enable implicit transactions in MySQL, set the autocommit mode to 0:

Example:

```
INSERT INTO Customers (Name) VALUES ('Alice'); -- Implicit transaction: Automatically committed if successful
```

```
UPDATE Products SET Price = 10 WHERE ID = 1; -- Implicit transaction: Automatically committed if successful
```

9. Explicit Transactions

1. Definition:

An **explicit transaction** is one where the transaction

boundaries are manually defined using START TRANSACTION, COMMIT, and ROLLBACK.

2. Characteristics:

- You explicitly define the start and end of a transaction.
- Offers precise control over transactional operations.
- Commonly used in complex operations where multiple statements need to be grouped together.

3. Commands:

- START TRANSACTION: Begins a new transaction.
- COMMIT: Saves changes permanently.
- ROLLBACK: Reverts changes to the last commit or savepoint.

Example:

```
START TRANSACTION; -- Explicitly begin a transaction

UPDATE accounts SET balance = balance - 500 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 500 WHERE account_id = 2;

COMMIT; -- Save the changes
```

Rollback Example:

```
START TRANSACTION;

UPDATE inventory SET stock = stock - 10 WHERE product_id = 101;

-- Simulate an error
ROLLBACK; -- Undo all changes
```

10. Key Differences

Feature	Implicit Transactions	Explicit Transactions
Start of Transaction	Automatically starts for certain commands	Manually started with START TRANSACTION
Commit/Rollback	Must be explicitly committed or rolled back	Must be explicitly committed or rolled back
Control	Limited control	Full control over transaction boundaries
Auto commit Dependency	Requires SET AUTOCOMMIT = 0	Independent of auto commit setting
Use Cases	Simple, single-statement operations	Complex, multi-statement operations requiring atomicity

3.Transaction Log Management

The **transaction log** in MySQL, commonly referred to as the **binary log (binlog)**, is a crucial component for managing and tracking changes to the database. It records all changes made to the database, ensuring data integrity, recovery, and replication.

11. What is a Transaction Log?

- A **transaction log** keeps a record of all modifications made to the database during transactions.
- It enables:
 1. **Data Recovery:** Restores the database to a consistent state in case of failures.
 2. **Replication:** Transfers changes to replicas in a master-slave setup.
 3. **Auditing:** Tracks changes for auditing purposes.

In MySQL, the **binary log** serves as the primary mechanism for transaction logging.

12. Key Components of MySQL Transaction Log Management

4. 1. Binary Log (Binlog)

- The binlog is used to log all SQL statements that modify data (e.g., INSERT, UPDATE, DELETE).
- It is also used for:
 - Point-in-time recovery.
 - Replication between master and replica servers.
- To enable the binary log, include the following in the MySQL configuration file (my.cnf or my.ini):

ini

Copy code

```
[mysqld]
```

```
log_bin = /path/to/binlog
```

5. 2. InnoDB Undo Logs

- The **undo logs** are internal logs used by the InnoDB storage engine to track changes for transactions that are not yet committed.
- They enable features like **rollback** and **MVCC (Multiversion Concurrency Control)**.

6. 3. Redo Logs

- **Redo logs** are used to ensure durability by recording all changes made to the data for recovery purposes in case of a crash.
 - They are stored in ib_logfile0, ib_logfile1, etc.
-

13. Configuring Transaction Logs

7. Binary Log Configuration

- To enable and configure the binary log:

ini

Copy code

[mysqld]

log_bin = /var/log/mysql/mysql-bin.log

binlog_format = ROW

expire_logs_days = 7

max_binlog_size = 100M

- log_bin: Enables binary logging.
- binlog_format: Specifies the logging format (ROW, STATEMENT, or MIXED).
- expire_logs_days: Sets the number of days to retain the binary logs.
- max_binlog_size: Limits the size of each binary log file.

8. InnoDB Log Configuration

- To configure redo and undo logs:

ini

Copy code

[mysqld]

innodb_log_file_size = 256M

innodb_log_buffer_size = 8M

innodb_undo_logs = 128

14. Managing and Monitoring Transaction Logs

9. 1. Checking Binary Logs

- List available binary logs:

sql

Copy code

SHOW BINARY LOGS;

- View the binary log file in use:

sql

Copy code

SHOW MASTER STATUS;

10. 2. Purging Old Logs

- To delete old binary logs:

sql

Copy code

```
PURGE BINARY LOGS TO 'mysql-bin.000010';
```

- To purge logs older than a specific date:

sql

Copy code

```
PURGE BINARY LOGS BEFORE '2024-01-01 00:00:00';
```

11. 3. Viewing Log Contents

- Use mysqlbinlog to view the contents of a binary log:

bash

Copy code

```
mysqlbinlog /path/to/mysql-bin.000001
```

12. 4. Disabling Binary Logs Temporarily

- You can disable binary logging for a specific session:

sql

Copy code

```
SET SESSION sql_log_bin = 0;
```

15. Best Practices for Transaction Log Management

1. Enable Binary Logging:

- Essential for recovery, replication, and auditing.

2. Optimize Log Sizes:

- Set appropriate sizes for binary logs and InnoDB logs to balance performance and disk usage.

3. Purge Old Logs:

- Regularly clean up logs to prevent disk space issues. Use `expire_logs_days` for automatic purging.

4. **Monitor Log Health:**

- Regularly monitor binary and InnoDB logs for corruption or excessive growth.

5. **Secure Logs:**

- Restrict access to transaction logs to prevent unauthorized access.
-

16. Recovery with Transaction Logs

13. Point-in-Time Recovery

1. Restore the latest backup.
2. Replay the binary logs to apply changes since the backup:

bash

Copy code

```
mysqlbinlog /var/log/mysql/mysql-bin.000001 | mysql -u username -p
```

17. DCL - Data Control Language

1. User and Permission Management

User and permission management (also known as Identity and Access Management or IAM) is a critical aspect of computer security and database administration. It involves controlling **who has access to what resources within a system**. This ensures that only authorized users can perform specific actions, protecting sensitive data and maintaining system integrity. MySQL provides robust tools to create, manage, and secure users and assign fine-grained permissions.

Key Concepts:

- **Users:** Individuals or entities that interact with the system. Each user typically has a unique identifier (username) and authentication credentials (password, biometric data, etc.).

- **Authentication:** The process of **verifying a user's identity**. This confirms that the user is who they claim to be. Common authentication methods include:
 - **Password-based authentication:** Using usernames and passwords.
 - **Multi-factor authentication (MFA):** Requiring multiple authentication factors (e.g., password and a one-time code from a mobile app).
 - **Biometric authentication:** Using fingerprints, facial recognition, or other biometric data.
- **Authorization:** The process of determining **what a user is allowed to do after they have been authenticated**. This involves checking the user's permissions or privileges.
- **Permissions/Privileges:** Specific actions that a user is allowed to perform on a resource (e.g., read, write, execute, delete).
- **Roles:** A collection of permissions that are grouped together and assigned to users. Roles simplify permission management by allowing administrators to assign permissions based on job functions or responsibilities.

Models for User and Permission Management:

- **Access Control Lists (ACLs):** Each resource has a list of users and their associated permissions. ACLs can become complex to manage in large systems.
- **Role-Based Access Control (RBAC):** Permissions are assigned to roles, and users are assigned to roles. This simplifies management and provides a more scalable approach.
- **Attribute-Based Access Control (ABAC):** Permissions are based on attributes of the user, the resource, and the environment. This provides fine-grained control and flexibility.

Implementation in Databases:

Most database management systems (DBMS) provide built-in mechanisms for user and permission management. These typically include:

- **Creating and managing user accounts:** Defining usernames, passwords, and other user properties.
- **Granting and revoking privileges:** Assigning specific permissions to users or roles on database objects (tables, views, stored procedures, etc.).
- **Defining roles:** Creating roles and assigning permissions to them.
- **Authentication mechanisms:** Supporting various authentication methods.

Example (Conceptual):

Imagine an e-commerce website with the following roles:

- **Administrator:** Has full access to all resources.
- **Manager:** Can manage products, orders, and customer data.
- **Customer:** Can view products, place orders, and manage their own account information.

The permission management system would:

1. **Authenticate** users when they log in.
2. **Authorize** users based on their assigned roles. For example:
 - An administrator would be granted all permissions.
 - A manager would be granted permissions to create, update, and delete products and orders, but not to manage user accounts.
 - A customer would be granted permissions to view products, place orders, and update their own profile, but not to access other customers' information or manage products.

Best Practices:

- **Principle of Least Privilege:** Grant users only the minimum necessary permissions to perform their tasks.
- **Regular Audits:** Regularly review user accounts and permissions to ensure they are still appropriate.
- **Strong Passwords:** Enforce strong password policies.

- **Multi-Factor Authentication:** Implement MFA for enhanced security.
 - **Role-Based Access Control:** Use RBAC to simplify permission management.
 - **Centralized Management:** Use a centralized system for managing users and permissions across multiple applications and systems.
-

1. User Management

1.1. Create a User

To create a new user:

```
CREATE USER 'username'@'host' IDENTIFIED BY 'password';
```

- **username:** The name of the user.
- **host:** The host from which the user can connect (e.g., 'localhost' or % for any host).
- **password:** The user's password.

Example:

```
CREATE USER 'john_doe'@'localhost' IDENTIFIED BY 'secure_password';
```

1.2. View Existing Users

List all users in the database:

```
SELECT User, Host FROM mysql.user;
```

1.3. Update a User's Password

```
ALTER USER 'username'@'host' IDENTIFIED BY 'new_password';
```

Example:

```
ALTER USER 'john_doe'@'localhost' IDENTIFIED BY 'new_secure_password';
```

1.4. Drop (Delete) a User

```
DROP USER 'username'@'host';
```

Example:

```
DROP USER 'john_doe'@'localhost';
```

2. Permission Management

Permissions (also called privileges) control what a user can do in the database.

2.1. Grant Permissions

```
GRANT privileges ON database.table TO 'username'@'host';
```

- **privileges:** The specific permissions to grant (e.g., SELECT, INSERT, ALL PRIVILEGES).
- **database.table:** The scope of the permissions. Use *.* for all databases and tables.

Example:

```
GRANT SELECT, INSERT ON university.students TO 'john_doe'@'localhost';
```

To grant all permissions:

```
GRANT ALL PRIVILEGES ON *.* TO 'admin_user'@'%';
```

2.2. View User Permissions

```
SHOW GRANTS FOR 'username'@'host';
```

Example:

```
SHOW GRANTS FOR 'john_doe'@'localhost';
```

2.3. Revoke Permissions

```
REVOKE privileges ON database.table FROM 'username'@'host';
```

Example:

```
REVOKE INSERT ON university.students FROM 'john_doe'@'localhost';
```

3. Common Permissions in MySQL

Permission	Description
SELECT	Allows reading data from tables.
INSERT	Allows adding new rows to tables.
UPDATE	Allows modifying existing rows in tables.
DELETE	Allows removing rows from tables.
CREATE	Allows creating new databases and tables.
DROP	Allows deleting databases and tables.
GRANT OPTION	Allows granting or revoking privileges to others.
ALL PRIVILEGES	Grants all permissions for the specified scope.

4. Manage Superuser (Root) Access

The root user in MySQL has full control over all databases and tables. Be cautious with its permissions.

1. Secure the Root Account

After installation:

```
ALTER USER 'root'@'localhost' IDENTIFIED BY 'secure_root_password';
```

2. Limit Root Access

Restrict root login to localhost:

```
UPDATE mysql.user SET Host = 'localhost' WHERE User = 'root';  
FLUSH PRIVILEGES;
```

5. Best Practices for User and Permission Management

1. Use least privilege:

- Grant users only the permissions they need.

2. Avoid using root:

- Create specific users for different applications and tasks.

3. Use strong passwords:

- Ensure all users have secure, non-trivial passwords.

4. Regularly audit users:

- Periodically review and update user privileges.

5. Restrict hosts:

- Allow access only from specific trusted IPs or domains.

6. Revoke unused users:

- Delete users that are no longer needed.

17. Delete vs Truncate

	Delete	Truncate
1	It is possible to delete only particular record or all the records	Truncate can only delete all the records, it is not possible to delete particular record
2	Delete operation is slower than Truncate	Truncate Operation is very faster
3	Deleted recorded can be rolled back.	With Truncate command, it is not possible to roll back the records.

18. Drop vs Truncate

S.N O	DROP	TRUNCATE
1.	The DROP command is used to remove table definition and its contents.	Whereas the TRUNCATE command is used to delete all the rows from the table.
2.	In the DROP command, table space is freed from memory.	While the TRUNCATE command does not free the table space from memory.
3.	DROP is a DDL(Data Definition Language) command.	Whereas the TRUNCATE is also a DDL(Data Definition Language) command.
4.	In the DROP command, view of table does not exist.	While in this command, view of table exist.
5.	In the DROP command, integrity constraints will be removed.	While in this command, integrity constraints will not be removed.
6.	In the DROP command, undo space is not used.	While in this command, undo space is used but less than DELETE.
7.	The DROP command is quick to perform but gives rise to complications.	While this command is faster than DROP.

19. Constraints:

SQL constraints are used **to specify rules** for data in a table.

1. Unique

Will not allow the duplicate values.

```
CREATE TABLE users (  
    user_id INT PRIMARY KEY,  
    username VARCHAR(50) UNIQUE  
);
```

2. Not Null

The column will not have the null values (it means column should have some value)

```
CREATE TABLE employees (  
    emp_id INT PRIMARY KEY,  
    name VARCHAR(100) NOT NULL,  
    email VARCHAR(100)  
);
```

3. Primary Key

Combination of both Unique and Not Null

```
CREATE TABLE orders (  
    order_id INT PRIMARY KEY,  
    order_date DATE NOT NULL  
);
```

Composite Key:

```
sql

CREATE TABLE order_details (
    order_id INT,
    product_id INT,
    PRIMARY KEY (order_id, product_id)
);
```

4. Foreign Key

The FOREIGN KEY constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(100)
);

CREATE TABLE salaries (
    salary_id INT PRIMARY KEY,
    emp_id INT,
    amount DECIMAL(10, 2),
    FOREIGN KEY (emp_id) REFERENCES employees(emp_id)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

5. Check

Ensures that all values in a column satisfy a specific condition

```
CREATE TABLE accounts (
    account_id INT PRIMARY KEY,
    balance DECIMAL(10, 2) CHECK (balance >= 0)
);
```

6. Default

Specifies a default value for a column if no value is provided during insertion.

```
CREATE TABLE products (  
    product_id INT PRIMARY KEY,  
    price DECIMAL(10, 2) DEFAULT 0.0  
);
```

Adding Constraints to Existing Tables

Adding NOT NULL :


```
sql  
  
ALTER TABLE employees MODIFY name VARCHAR(100) NOT NULL;
```

Adding UNIQUE :

```
sql  
  
ALTER TABLE employees ADD CONSTRAINT unique_email UNIQUE (email);
```


Adding PRIMARY KEY :

```
sql  
  
ALTER TABLE employees ADD PRIMARY KEY (emp_id);
```

 Copy code


Adding FOREIGN KEY :

```
sql  
  
ALTER TABLE salaries ADD CONSTRAINT fk_emp FOREIGN KEY (emp_id) REFERENCES employees(emp_i
```

 Copy code

Adding CHECK :

```
sql  
  
ALTER TABLE accounts ADD CONSTRAINT chk_balance CHECK (balance >= 0);
```

 Copy code



Dropping Constraints

Drop NOT NULL :

sql

```
ALTER TABLE employees MODIFY name VARCHAR(100) NULL;
```

Drop UNIQUE :

sql

```
ALTER TABLE employees DROP INDEX unique_email;
```

Drop PRIMARY KEY :

sql

```
ALTER TABLE employees DROP PRIMARY KEY;
```

Drop FOREIGN KEY :

sql

```
ALTER TABLE salaries DROP FOREIGN KEY fk_emp;
```

Drop CHECK :

sql

```
ALTER TABLE accounts DROP CONSTRAINT chk_balance;
```

20. Where clause:

1. Relational operators:

<, <=, >, >=, !=

2. Logical Operators:

And, or and Not

3. Like and Not like

4. In, not in

5. Between, Not between

6. Exists and not Exists

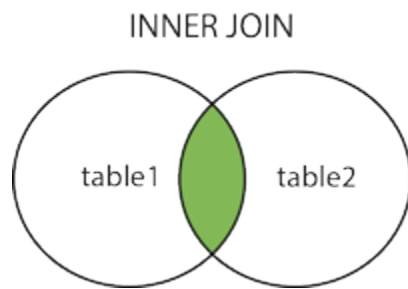
7. Wildcards

21. Join

Joining the two or more than the tables and retrieve required no of columns from the tables.

1. Inner Join

The **INNER JOIN** is one of the most commonly used joins in SQL. It retrieves records that have matching values in both tables being joined. If a row in one table does not have a corresponding row in the other table, it will not be included in the result.



Syntax:

```
SELECT column_list
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

- **table1 and table2:** The tables being joined.
- **ON:** Specifies the condition for matching rows between the tables.

2. Key Characteristics

- Returns rows where there is a match in both tables.
- Rows without matches in either table are excluded from the result.

Example

Tables:

Result Grid					
Filter Rows:					
	Emp_ID	Name	Salary	Address	Dept
▶	122	Nikhil	53934.43	12th street	Mech
	123	Senthil	99999.99	11th Street	CSE
	125	Nikhil	45324.67	NULL	ECE
	126	Gowth	85324.67	1st Street	CSE
	127	Arun	65324.67	2nd Street	CSE
	128	Bala	31563.87	3rd Street	IT
	130	Kumar	63272.43	10th Street	Mech
	231	Raju	371563.87	3rd Street	ECE

Result Grid			
Filter Rows:			
	Name	BankName	AccNo
▶	Senthil	Axis Bank	213422
	Ashok	SBI Bank	543664
	Kumar	ICICI Bank	565745576
	Sathis	HDFC Bank	32453637
	Aswini	Indian Bank	192726
	Senthil	HDFC Bank	234787
	Senthil	ICICI Bank	3323726

Another Example:

employees:

employee_id	name	department_id
1	Alice	10
2	Bob	20
3	Charlie	NULL

departments:

department_id	department_name
10	HR
20	IT
30	Finance

Query with INNER JOIN:

```
SELECT
    employees.name,
    departments.department_name
FROM
    employees
INNER JOIN
    departments
ON
    employees.department_id = departments.department_id;
```

Result:

name	department_name
Alice	HR

name	department_name
Bob	IT

- **Explanation:**

- Alice and Bob have matching department_id values in the departments table.
- Charlie does not have a matching department_id, so their row is excluded.

3. Alias for Simplicity

You can use table aliases to make the query easier to read:

```
SELECT
    e.name,
    d.department_name
FROM
    employees e
INNER JOIN
    departments d
ON
    e.department_id = d.department_id;
```

4. Joining Multiple Tables

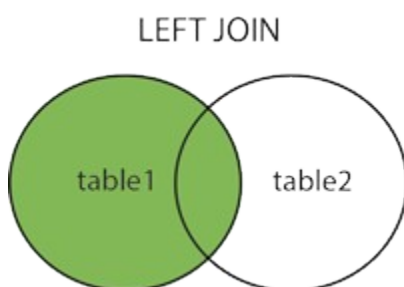
You can join more than two tables using multiple INNER JOIN clauses:

```
SELECT
    e.name,
    d.department_name,
    m.manager_name
FROM
    employees e
INNER JOIN
    departments d
ON
    e.department_id = d.department_id
INNER JOIN
    managers m
ON
    d.department_id = m.department_id;
```

5. Key Use Cases

1. Retrieving related data from multiple tables.
2. Filtering data to include only rows with matching relationships.
3. Building complex queries involving multiple join

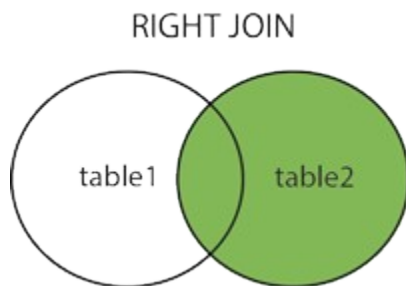
6. Left Join



Syntax:

Example:

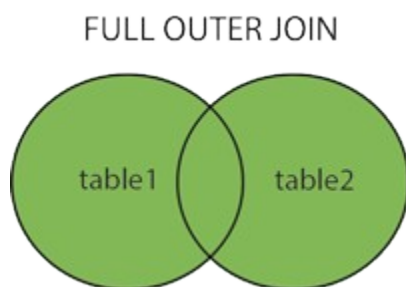
7. Right Join



Syntax:

Example:

8. Full Join / Outer Join



Syntax:

Example:

9. Cross Join

The **CROSS JOIN** is used to combine every row from one table with every row from another table, resulting in a Cartesian product of the two tables. Unlike other types of joins, it does not require any condition to match rows.

Syntax

```
SELECT column_list  
FROM table1  
CROSS JOIN table2;
```

- **table1 and table2:** The tables being combined.

10. Key Characteristics

- Produces a **Cartesian product**, which means the number of rows in the result set is equal to the product of the number of rows in the two tables.
- Typically used for scenarios where all combinations of rows are needed.
- Can generate a large result set if the tables are large.

Example

Tables:

products:

product_id	product_name
1	Laptop
2	Smartphone

regions:

region_id	region_name
101	North

region_id	region_name
	America
102	Europe

Query with CROSS JOIN:

```
SELECT
    products.product_name,
    regions.region_name
FROM
    products
CROSS JOIN
    regions;
```

Result:

product_name	region_name
Laptop	North America
Laptop	Europe
Smartphone	North America
Smartphone	Europe

- **Explanation:**

- Each row in the products table is combined with every row in the regions table.

11. Implicit CROSS JOIN

If no join condition is specified, a **CROSS JOIN** is implied by default:

```
SELECT
    products.product_name,
    regions.region_name
FROM
    products, regions;
```

This also produces a Cartesian product.

12. Use Cases

1. Testing and Debugging:

- Used to generate all possible combinations of rows for testing queries or understanding data relationships.

2. Special Scenarios:

- Calculating combinations (e.g., generating price combinations for products in different regions).
- Creating datasets for statistical analysis.

3. Simulating Combinations:

- For example, pairing each salesperson with every product.

13. Key Considerations

- **Size of Result Set:** Be cautious when using CROSS JOIN with large tables as it can generate an enormous number of rows.
 - Number of rows in the result = (Rows in Table 1) × (Rows in Table 2).
- Usually, a CROSS JOIN is followed by a filtering condition using a WHERE clause to limit the results.

14. Example with Filtering

```

SELECT
    products.product_name,
    regions.region_name
FROM
    products
CROSS JOIN
    regions
WHERE
    regions.region_name = 'Europe';

```

Result:

product_name	region_name
Laptop	Europe
Smartphone	Europe

- **Explanation:** The result is limited to combinations where the region is Europe.

15. Self Join

Syntax:

Example:

16. Equi Join

Syntax:

Example:

17. What is the difference between Equi Join and Inner Join in SQL?

An equijoin is a join with a join condition containing an equality operator. An equijoin returns only the rows that have equivalent values for the specified columns.

An inner join is a join of two or more tables that returns only those rows (compared using a comparison operator) that satisfy the join condition.

Pictorial representation : EQUI JOIN Vs. INNER JOIN

EQUI JOIN

table_a

ID	COL1
2	A2
5	A5
3	A3
1	-
4	A4

table_b

ID	COL2
5	B5
1	B1
3	-
6	B6
2	B2
5	C5

no equality
in table_b

no equality
in table_a

VS.

INNER JOIN

table_a

ID	COL1
2	A2
5	A5
3	A3
1	-
4	A4

table_b

ID	COL2
5	B5
1	B1
3	-
6	B6
2	B2
5	C5

no matches
in table_b

no matches
in table_a

```
SQL> SELECT * FROM TABLE_A
2 JOIN TABLE_B
3 ON TABLE_A.ID=TABLE_B.ID;
```

ID	COL1	ID	COL2
5	A5	5	B5
1	-	1	B1
3	A3	3	-
2	A2	2	B2
5	A5	5	C5

```
SQL> SELECT * FROM TABLE_A
2 JOIN TABLE_B
3 ON TABLE_A.ID=TABLE_B.ID;
```

ID	COL1	ID	COL2
5	A5	5	B5
1	-	1	B1
3	A3	3	-
2	A2	2	B2
5	A5	5	C5

22. Subqueries

Noncorrelated and Correlated Subqueries

Subqueries can be categorized into two types:

- A *noncorrelated* (simple) subquery obtains its results **independently** of its containing (outer) statement.
- A *correlated* subquery requires values from its outer query in order to execute the inner query.

1.Non-correlated Subqueries

A noncorrelated subquery executes independently of the outer query. The subquery executes first, and then passes its results to the outer query, For example:

=> SELECT name, street, city, state FROM addresses WHERE state IN (**SELECT state FROM states**);

Vertica executes this query as follows:

1. Executes the subquery SELECT state FROM states (in bold).
2. Passes the subquery results to the outer query.

A query's WHERE and HAVING clauses can specify noncorrelated subqueries if the subquery resolves to a single row, as shown below:

In WHERE clause

=> SELECT COUNT(*) FROM SubQ1 WHERE SubQ1.a = (SELECT y from SubQ2);

In HAVING clause

=> SELECT COUNT(*) FROM SubQ1 GROUP BY SubQ1.a HAVING SubQ1.a = (SubQ1.a & (SELECT y from SubQ2))

2. Correlated Subqueries

A correlated subquery typically obtains **values from its outer query before inner query executes**. When the subquery returns, it passes its results to the outer query.

You can use an outer join to obtain the same effect as a correlated subquery.

In the following example, the subquery needs values from the addresses.state column in the outer query:

Example:

=> SELECT name, street, city, state FROM addresses
WHERE EXISTS (SELECT * FROM states WHERE states.state = addresses.state);

Vertica executes this query as follows:

1. The query extracts and evaluates each addresses.state value in the outer subquery records.
2. Then the query—using the EXISTS predicate—checks the addresses in the inner (correlated) subquery.
3. Because it uses the EXISTS predicate, the query stops processing when it finds the first match.

When Vertica executes this query, it translates the full query into a JOIN WITH SIPS.

Example:

SELECT employee_number, name
FROM employees emp

```
WHERE salary > (SELECT AVG(salary)
FROM employees
WHERE department = emp.department);
```

In the above case, for each employee, the inner query calculates the average salary for their department.

23. Super Key vs Candidate Key

Super Key	Candidate Key
Can have one or more attributes, and may include extra, unnecessary columns	Is a minimal super key , meaning it contains no unnecessary columns
Every candidate key is a super key, but not every super key is a candidate key	All candidate keys are super keys , but not all super keys are candidate keys.
There can be many super keys, some of which may have redundant columns.	Candidate keys are the smallest possible set of attributes that uniquely identify a record
Example: {ID, Name}, {ID, Email}	Example: {ID}, {Email} if both are minimal

24. Scalar Functions

1. Numeric Functions

MySQL provides a wide range of numeric functions that allow you to perform mathematical calculations and manipulate numeric data. Here's a summary of commonly used numeric functions in MySQL:

1. Arithmetic Functions

Function	Description	Example	Result
ABS(x)	Returns the absolute value of	ABS(-10)	10

Function	Description	Example	Result
	x.		
CEIL(x) or CEILING(x)	Returns the smallest integer greater than or equal to x.	CEIL(4.3)	5
FLOOR(x)	Returns the largest integer less than or equal to x.	FLOOR(4.7)	4
ROUND(x, d)	Rounds x to d decimal places.	ROUND(123.456, 2)	123.46
SIGN(x)	Returns the sign of x (-1, 0, 1).	SIGN(-42)	-1
TRUNCATE(x, d)	Truncates x to d decimal places without rounding.	TRUNCATE(123.456, 2)	123.45

2. Mathematical Constants and Functions

Function	Description	Example	Result
PI()	Returns the value of π (pi).	PI()	3.141593
EXP(x)	Returns e^x (Euler's number raised to the power of x).	EXP(1)	2.718282
LOG(x)	Returns the natural logarithm of x.	LOG(2.718282)	1
LOG10(x)	Returns the base-10 logarithm of x.	LOG10(1000)	3
POW(x, y) or POWER(x, y)	Returns x raised to the power of y.	POW(2, 3)	8
SQRT(x)	Returns the square root of x.	SQRT(16)	4

3. Trigonometric Functions

Function	Description	Example	Result
SIN(x)	Returns the sine of x (in radians).	SIN(PI()/2)	1

Function	Description	Example	Result
COS(x)	Returns the cosine of x (in radians).	COS(PI())	-1
TAN(x)	Returns the tangent of x (in radians).	TAN(PI()/4)	1
ASIN(x)	Returns the arcsine of x (in radians).	ASIN(1)	1.5708 ($\pi/2$)
ACOS(x)	Returns the arccosine of x (in radians).	ACOS(-1)	3.141593 (π)
ATAN(x)	Returns the arctangent of x (in radians).	ATAN(1)	0.785398 ($\pi/4$)
DEGREES(x)	Converts x from radians to degrees.	DEGREES(PI()/2)	90
RADIANS(x)	Converts x from degrees to radians.	RADIANS(180)	3.141593 (π)

4. Random Numbers

Function	Description	Example	Result
RAND()	Returns a random float value between 0 and 1.	RAND()	0.548813 (varies)
RAND(x)	Returns a random float value seeded by x.	RAND(10)	Consistent result for the seed.

5. Bitwise Functions

Function	Description	Example	Result
BIT_AND(x, y)	Performs a bitwise AND operation on x and y.	BIT_AND(5,3)	1
BIT_OR(x, y)	Performs a bitwise OR operation on x and y.	BIT_OR(5,3)	7

Function	Description	Example	Result
BIT_XOR(x, y)	Performs a bitwise XOR operation on x and y.	BIT_XOR(5,3)	6

6. Numeric Type Conversion

Function	Description	Example	Result
CONV(x, from_base, to_base)	Converts number x from one base to another.	CONV(15, 10, 2)	1111
CAST(x AS UNSIGNED)	Converts x to an unsigned integer.	CAST(123.45 AS UNSIGNED)	123

Usage Example

```
SELECT
    ABS(-15) AS AbsoluteValue,
    ROUND(123.456, 2) AS RoundedValue,
    SIN(PI()/2) AS SineValue,
    RAND() AS RandomValue;
```

Result:

Result:

plaintext

```
+-----+-----+-----+-----+
| AbsoluteValue | RoundedValue | SineValue | RandomValue |
+-----+-----+-----+-----+
| 15           | 123.46       | 1.000000 | 0.548813    |
+-----+-----+-----+-----+
```

2.String Functions:

MySQL provides a variety of string functions to manipulate and process string data. Below is a comprehensive list of commonly used string functions:

1.String Manipulation Functions

Function	Description	Example	Result
CONCAT(str1, str2, ...)	Concatenates strings.	CONCAT('Hello', ' ', 'World')	Hello World
CONCAT_WS(separator, str1, str2, ...)	Concatenates strings with a specified separator.	CONCAT_WS('-', '2024', '12', '30')	2024-12-30
LEFT(str, length)	Returns the leftmost length characters of the string.	LEFT('MySQL', 3)	MyS
RIGHT(str, length)	Returns the rightmost length characters of the string.	RIGHT('MySQL', 3)	SQL
SUBSTRING(str, pos, len)	Extracts a substring starting at pos for len characters.	SUBSTRING('MySQL', 2, 3)	ySQ
TRIM(str) or TRIM([remstr] FROM str)	Removes leading/trailing spaces or specified characters from a string.	TRIM(' MySQL ')	MySQL
LTRIM(str)	Removes leading spaces.	LTRIM(' MySQL')	MySQL
RTRIM(str)	Removes trailing spaces.	RTRIM('MySQL ')	MySQL
REPLACE(str, from_str, to_str)	Replaces occurrences of from_str with to_str.	REPLACE('Hello World', 'World', 'SQL')	Hello SQL

Function	Description	Example	Result
INSERT(str, pos, len, newstr)	Inserts newstr into str starting at pos, replacing len characters.	INSERT('Hello', 2, 2, 'i')	Hiilo

2. String Case Conversion

Function	Description	Example	Result
UPPER(str)	Converts a string to uppercase.	UPPER('mysql')	MYSQL
LOWER(str)	Converts a string to lowercase.	LOWER('MySQL')	mysql
INITCAP(str) (Not native; use CONCAT)	Capitalizes the first letter of each word.	CONCAT(UPPER(LEFT('mysql',1)),LOWER(SUBSTRING('mysql',2)))	Mysql

3. String Length and Position

Function	Description	Example	Result
LENGTH(str)	Returns the byte length of a string.	LENGTH('MySQL')	5
CHAR_LENGTH(str)	Returns the character length of a string.	CHAR_LENGTH('MySQL')	5
LOCATE(substr, str, pos)	Returns the position of the first occurrence of substr.	LOCATE('S', 'MySQL')	3
POSITION(substr IN str)	Similar to LOCATE.	POSITION('S' IN 'MySQL')	3
INSTR(str, substr)	Returns the position of the first occurrence of substr.	INSTR('MySQL', 'y')	2

4. Padding and Repeating Strings

Function	Description	Example	Result
LPAD(str, len, padstr)	Left-pads the string str with padstr to length len.	LPAD('SQL', 5, '0')	00SQL
RPAD(str, len, padstr)	Right-pads the string str with padstr to length len.	RPAD('SQL', 5, '-')	SQL--
REPEAT(str, count)	Repeats a string count times.	REPEAT('My', 3)	MyMyMy

5. String Comparison

Function	Description	Example	Result
STRCMP(str1, str2)	Compares two strings lexicographically.	STRCMP('abc', 'abd')	-1
LIKE	Checks if a string matches a pattern.	'Hello' LIKE 'H%'	1 (TRUE)
NOT LIKE	Checks if a string does not match a pattern.	'Hello' NOT LIKE 'H%'	0 (FALSE)

6. String Encoding and Decoding

Function	Description	Example	Result
ASCII(str)	Returns the ASCII code of the first character in the string.	ASCII('A')	65
CHAR(N, ...)	Converts ASCII codes to characters.	CHAR(65, 66, 67)	ABC
HEX(str)	Returns a hexadecimal representation of the string.	HEX('ABC')	414243
UNHEX(hex_str)	Converts a hexadecimal string back to its original form.	UNHEX('414243')	ABC

Usage Example

```

SELECT
  CONCAT(UPPER(LEFT('mysql', 1)), LOWER(SUBSTRING('mysql', 2))) AS Capitalized,
  REPLACE('Hello World', 'World', 'MySQL') AS Replaced,
  LPAD('SQL', 5, '*') AS Padded,
  LENGTH('MySQL') AS Length,
  LOCATE('S', 'MySQL') AS Position;

```

Result:

```

+-----+-----+-----+-----+-----+
| Capitalized | Replaced | Padded | Length | Position |
+-----+-----+-----+-----+-----+
| Mysql      | Hello MySQL | **SQL  | 5      | 3        |
+-----+-----+-----+-----+-----+

```

3. Date Functions

MySQL provides a variety of date and time functions to handle, format, and manipulate date and time values. Here's a detailed list of commonly used date functions:

1. Current Date and Time

Function	Description	Example	Result
CURDATE()	Returns the current date.	CURDATE()	2024-12-30
CURRENT_DATE()	Synonym for CURDATE().	CURRENT_DATE()	2024-12-30
NOW()	Returns the current date and time.	NOW()	2024-12-30 14:25:36
CURRENT_TIMESTAMP()	Synonym for NOW().	CURRENT_TIMESTAMP()	2024-12-30 14:25:36
CURTIME()	Returns the current time.	CURTIME()	14:25:36
CURRENT_TIME()	Synonym for CURTIME().	CURRENT_TIME()	14:25:36

2. Extracting Parts of Dates

Function	Description	Example	Result
YEAR(date)	Extracts the year from a date.	YEAR('2024-12-30')	2024
MONTH(date)	Extracts the month (1-12) from a date.	MONTH('2024-12-30')	12
DAY(date) or DAYOFMONTH(date)	Extracts the day of the month (1-31).	DAY('2024-12-30')	30
HOUR(time)	Extracts the hour (0-23) from a time or datetime.	HOUR('14:25:36')	14
MINUTE(time)	Extracts the minute (0-59) from a time or datetime.	MINUTE('14:25:36')	25
SECOND(time)	Extracts the second (0-59) from a time or datetime.	SECOND('14:25:36')	36
DAYOFWEEK(date)	Returns the weekday index (1=Sunday, 7=Saturday).	DAYOFWEEK('2024-12-30')	2
DAYOFYEAR(date)	Returns the day of the year (1-366).	DAYOFYEAR('2024-12-30')	365
WEEK(date)	Returns the week number (0-53).	WEEK('2024-12-30')	52
QUARTER(date)	Returns the quarter (1-4) of the year.	QUARTER('2024-12-30')	4

3. Adding and Subtracting Dates

Function	Description	Example	Result
DATE_ADD(date, INTERVAL value unit)	Adds a time interval to a date.	DATE_ADD('2024-12-30', INTERVAL 10 DAY)	2024-01-09

Function	Description	Example	Result
DATE_SUB(date, INTERVAL value unit)	Subtracts a time interval from a date.	DATE_SUB('2024-12-30', INTERVAL 10 DAY)	2024-12-20
ADDDATE(date, INTERVAL value unit)	Synonym for DATE_ADD().	ADDDATE('2024-12-30', INTERVAL 1 MONTH)	2025-01-30
SUBDATE(date, INTERVAL value unit)	Synonym for DATE_SUB().	SUBDATE('2024-12-30', INTERVAL 1 MONTH)	2024-11-30

4. Formatting Dates

Function	Description	Example	Result
DATE_FORMAT(date, format)	Formats a date according to the given format.	DATE_FORMAT('2024-12-30', '%d-%b-%Y')	30-Dec-2024
STR_TO_DATE(str, format)	Converts a string to a date using the specified format.	STR_TO_DATE('30-12-2024', '%d-%m-%Y')	2024-12-30

5. Common Format Specifiers:

Specifier	Description	Example
%Y	Year (4 digits).	2024
%y	Year (2 digits).	24
%M	Full month name.	December
%b	Abbreviated month name.	Dec
%d	Day of the month (2 digits).	30
%H	Hour (24-hour)	14

Specifier	Description	Example
	format).	
%h	Hour (12-hour format).	02
%i	Minutes.	25
%s	Seconds.	36
%p	AM or PM.	PM

6. Calculating Differences

Function	Description	Example	Result
DATEDIFF(date1, date2)	Returns the difference in days between two dates.	DATEDIFF('2024-12-30', '2024-12-25')	5
TIMEDIFF(time1, time2)	Returns the difference between two times.	TIMEDIFF('14:25:36', '12:00:00')	02:25:36
TIMESTAMPDIFF(unit, datetime1, datetime2)	Returns the difference between two dates or times in the specified unit.	TIMESTAMPDIFF(DAY, '2024-12-25', '2024-12-30')	5

7. Other Date Functions

Function	Description	Example	Result
LAST_DAY(date)	Returns the last day of the month for the given date.	LAST_DAY('2024-12-15')	2024-12-31
MAKEDATE(year, dayofyear)	Creates a date from the year and day of the year.	MAKEDATE(2024, 365)	2024-12-30
MAKETIME(hour, minute, second)	Creates a time from the given values.	MAKETIME(14, 25, 36)	14:25:36

Usage Example

```
SELECT
    CURDATE() AS CurrentDate,
    DATE_FORMAT(CURDATE(), '%d-%b-%Y') AS FormattedDate,
    YEAR(CURDATE()) AS Year,
    MONTH(CURDATE()) AS Month,
    DAY(CURDATE()) AS Day,
    DATE_ADD(CURDATE(), INTERVAL 7 DAY) AS NextWeek,
    DATEDIFF('2024-12-31', CURDATE()) AS DaysLeft;
```

Result:

CurrentDate	FormattedDate	Year	Month	Day	NextWeek	DaysLeft
2024-12-30	30-Dec-2024	2024	12	30	2024-01-06	1

25. Windows Function

Window functions in MySQL (introduced in MySQL 8.0) perform calculations across a set of rows that are related to the current row. Unlike aggregate functions (like SUM(), AVG(), COUNT()), which collapse multiple rows into a single output row, window functions retain the original rows while adding calculated values based on a "window" of related rows.

Key Concepts

- **Window:** The "window" refers to the set of rows on which the function operates. This window is defined relative to the current row.
- **OVER() Clause:** This clause is essential for window functions. It defines the window by specifying how the rows are partitioned and ordered.
- **PARTITION BY:** This clause divides the rows into partitions or groups. The window function is applied to each partition independently.

- **ORDER BY:** This clause specifies the order of rows within each partition. This is crucial for functions that depend on the order of rows, like ROW_NUMBER() or LAG().

Types of Window Functions

1. **Aggregate Window Functions:** These functions perform aggregate calculations (like SUM(), AVG(), COUNT(), MIN(), MAX()) over a window of rows.
 - Example: Calculate the running total of sales for each day.
2. **Ranking Window Functions:** These functions assign a rank to each row within a partition based on a specified order.
 - **ROW_NUMBER():** Assigns a unique sequential integer to each row within a partition.
 - **RANK():** Assigns a rank to each row within a partition based on the specified order. Rows with the same value receive the same rank, and the next rank is skipped.
 - **DENSE_RANK():** Similar to RANK(), but it doesn't skip ranks.
 - **NTILE(n):** Divides the rows within a partition into n approximately equal groups and assigns a group number to each row.
3. **Value Window Functions:** These functions access values from other rows within the window.
 - **LAG(expression, offset, default):** Accesses data from a previous row within the partition. offset specifies how many rows back to look, and default provides a value if there's no previous row.
 - **LEAD(expression, offset, default):** Accesses data from a subsequent row within the partition.
 - **FIRST_VALUE(column):** Returns the first value of a column in the window.
 - **LAST_VALUE(column):** Returns the last value of a column in the window.

Syntax:

```
function_name([arguments]) OVER (  
    [PARTITION BY column_name]  
    [ORDER BY column_name]  
    [FRAME_SPECIFICATION]  
)
```

- **function_name**: The window function to apply (e.g., ROW_NUMBER(), RANK(), SUM(), etc.).
- **OVER**: Specifies the window for the function.
- **PARTITION BY**: Divides the result set into partitions to apply the window function independently on each partition.
- **ORDER BY**: Orders the rows within each partition.
- **FRAME_SPECIFICATION**: Defines the subset of rows for the calculation (optional).

Example:

```
SELECT employee_id, department_id, salary,  
       ROW_NUMBER() OVER (PARTITION BY department_id ORDER BY salary DESC) AS rank  
FROM employees;
```

- Assigns a rank to each employee within their department based on salary (highest first).
-

2. Using SUM() with a Window

```
SELECT department_id, employee_id, salary,  
       SUM(salary) OVER (PARTITION BY department_id) AS total_salary  
FROM employees;
```

- Calculates the total salary for each department.
-

3. Using RANK()

```
SELECT employee_id, department_id, salary,  
       RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS rank  
FROM employees;
```


- Similar to ROW_NUMBER(), but assigns the same rank to duplicate salaries, leaving gaps.
-

4. Using DENSE_RANK()

```
SELECT EmployeeID, Department, Salary,  
       DENSE_RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS SalaryRa  
FROM Employees;
```

The **DENSE_RANK()** function is that assigns a rank to rows within a result set, with no gaps in the ranking values when there are ties. This is different from the RANK() function, which skips ranks after ties

5. Using NTILE()

```
SELECT employee_id, salary,  
       NTILE(4) OVER (ORDER BY salary DESC) AS quartile  
FROM employees;
```

- Divides employees into four groups (quartiles) based on salary.
-

6. Using LEAD() and LAG()

```
SELECT employee_id, salary,  
       LAG(salary, 1) OVER (ORDER BY salary) AS previous_salary,  
       LEAD(salary, 1) OVER (ORDER BY salary) AS next_salary  
FROM employees;
```

- Retrieves the salary of the previous and next employees in the order of salary.
-

7. Using FIRST_VALUE() and LAST_VALUE()

```
SELECT department_id, employee_id, salary,  
       FIRST_VALUE(salary) OVER (PARTITION BY department_id  
ORDER BY salary DESC) AS highest_salary,
```

```
LAST_VALUE(salary) OVER (PARTITION BY department_id  
ORDER BY salary DESC ROWS BETWEEN UNBOUNDED PRECEDING  
AND UNBOUNDED FOLLOWING) AS lowest_salary
```

FROM employees;

- Finds the highest and lowest salary within each department.
 - ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING defines a **window that includes all rows in the partition** (or the entire result set), which is essential for certain window function use cases, especially when you need to access the true last value within a set of data.
-

4. Key Notes

1. OVER Clause:

- Without PARTITION BY: Applies the function to the entire result set.
- Without ORDER BY: The order is undefined for the function.

2. Default Frame:

- If no frame is specified, the default is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

3. Performance:

- Window functions can be resource-intensive on large datasets. Optimize queries and indexes accordingly.
-

5. When to Use Window Functions

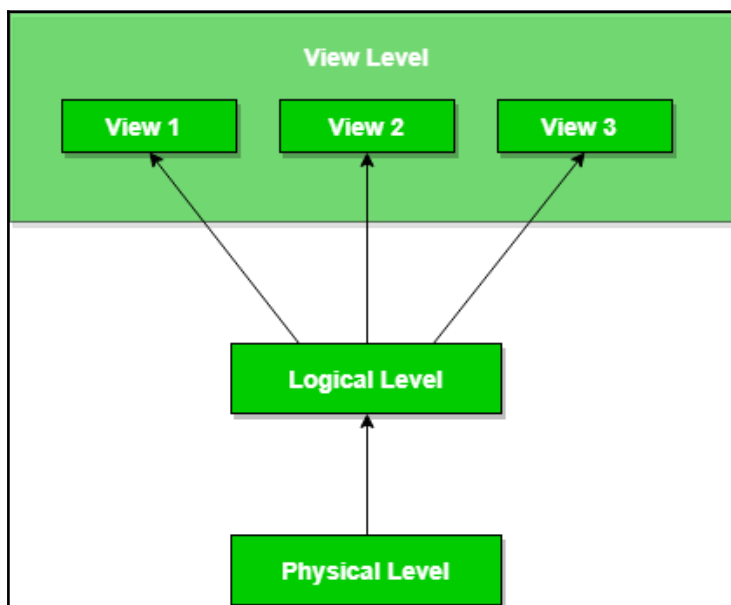
- **Ranking and Pagination:** Generate row numbers for pagination or ranking.
- **Cumulative Calculations:** Perform running totals or averages.
- **Comparing Rows:** Use LEAD() or LAG() to compare data in adjacent rows.
- **Analyzing Data:** Derive insights like top performers in each group.

Benefits of Window Functions

- **Simplified Queries:** Window functions can simplify complex queries that would otherwise require subqueries or self-joins.
- **Improved Performance:** In many cases, window functions can be more efficient than equivalent queries using subqueries or joins.
- **Enhanced Data Analysis:** Window functions enable advanced data analysis tasks like calculating running totals, moving averages, and rankings.

Window functions are a valuable tool in SQL for performing complex calculations and analysis on data. They provide a concise and efficient way to access and compare data from multiple rows without losing the individual row context.

26. 3 levels of abstraction



27. PreparedStatement

The PreparedStatement interface is a sub interface of Statement. It is used to execute parameterized query.

```
String sql="insert into emp values(?,?,?)";
```

Why use PreparedStatement?

Improves performance: The performance of the application will be faster if you use PreparedStatement interface because **query is compiled only once**.

An important advantage of PreparedStatements is that they **prevent SQL injection attacks**.

Try to find sample SQL Injection attacks

Metadata Objects:

The Question marks in PreparedStatement is called Metadata Objects

28. CallableStatement

CallableStatement interface is used to call the **stored procedures and functions**

29. StoredProcedure vs Function

Stored Procedure	Function
is used to perform business logic.	is used to perform calculation.
must not have the return type.	must have the return type.
may return 0 or more values.	may return only one values.
We can call functions from the procedure.	Procedure cannot be called from function.
Procedure supports input and output parameters.	Function supports only input parameter.
Exception handling using try/catch block can be used in stored procedures.	Exception handling using try/catch can't be used in user defined functions.

30. Transactions

Transaction represents **a single unit of work**

The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, Isolation and Durability.

Atomicity means either all successful or none.

Consistency ensures bringing the database from one consistent state to another consistent state.

Isolation ensures that transaction is isolated from other transaction.

Durability means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.


31. Advantage of Transaction Mangement

fast performance It makes the performance fast because database is hit at the time of commit.

32. Common Table Expression


A Common Table Expression (CTE) in SQL is a temporary result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. CTEs are often used to simplify complex queries, improve readability, and organize query logic.

sql

 Copy code

```
WITH cte_name (column1, column2, ...) AS (  
    -- CTE query definition  
    SELECT ...  
)  
-- Main query  
SELECT ...  
FROM cte_name
```

sql

 Copy code

```
WITH DepartmentSalary AS (  
    SELECT  
        department_id,  
        AVG(salary) AS average_salary  
    FROM  
        employees  
    GROUP BY  
        department_id  
)  
SELECT  
    e.first_name,  
    e.last_name,  
    e.department_id,  
    ds.average_salary  
FROM  
    employees e  
JOIN  
    DepartmentSalary ds  
ON  
    e.department_id = ds.department_id;
```

33. Views

In PostgreSQL (and many other relational database management systems), a view is a virtual table that is defined by a SQL query. Unlike a physical table, a view does not store data itself but rather provides a way to present data from one or more tables in a specific format. Views can simplify complex queries, encapsulate business logic, and enhance security by restricting access to specific data.

34. Sequence

a sequence is a database object used to generate a sequence of unique integer values. Sequences are often used to generate unique primary key values for tables automatically

sql

```
CREATE SEQUENCE sequence_name
  [ INCREMENT BY increment ]
  [ MINVALUE minvalue ]
  [ MAXVALUE maxvalue ]
  [ START WITH start ]
  [ CYCLE | NO CYCLE ]
  [ CACHE cache_size ];
```

- `sequence_name`: The name of the sequence.
- `INCREMENT BY`: Optional. The increment value (default is 1).
- `MINVALUE`: Optional. The minimum value the sequence can generate (default is `1`).
- `MAXVALUE`: Optional. The maximum value the sequence can generate (default is $2^{63} - 1$).
- `START WITH`: Optional. The starting value of the sequence (default is `1`).
- `CYCLE`: Optional. If specified, restarts the sequence from `MINVALUE` or `START WITH` after reaching `MAXVALUE`. If not specified and the sequence reaches `MAXVALUE`, further requests for `nextval` will return an error.



sql

```
CREATE SEQUENCE employee_id_seq
  START WITH 100
  INCREMENT BY 1
  NO MINVALUE
  NO MAXVALUE
  CACHE 10;
```

sql

```
SELECT nextval('employee_id_seq');
```

sql

```
DROP SEQUENCE sequence_name;
```

sql

```
CREATE TABLE employees (  
    employee_id INTEGER PRIMARY KEY DEFAULT nextval('employee_id_seq'),  
    first_name VARCHAR(50) NOT NULL,  
    last_name VARCHAR(50) NOT NULL,  
    hire_date DATE NOT NULL  
);
```

Index

an index is a database object that enhances the speed of data retrieval operations on a table at the cost of additional storage space and overhead during data modification operations (inserts, updates, and deletes). Indexes are essential for optimizing query performance, especially for large datasets.

Types of Indexes

□ **B-tree Index:**

- The default and most common type of index.

- Suitable for equality and range queries.
- Supports =, <, <=, >, >=, and BETWEEN operators.

□ **Hash Index:**

- Suitable for equality comparisons (=).
- Not as commonly used because it does not support range queries.

□ **GIN (Generalized Inverted Index):**

- Suitable for indexing composite values, like arrays, JSONB, and full-text search.
- Efficient for containment queries (e.g., checking if an array contains a specific value).

□ **GiST (Generalized Search Tree):**

- Suitable for complex data types, such as geometric data types and full-text search.
- Supports various types of queries depending on the operator class.

□ **SP-GiST (Space-Partitioned Generalized Search Tree):**

- Suitable for data that can be divided into non-overlapping partitions.
- Useful for certain types of geometric and text search operations.

□ **BRIN (Block Range Index):**

- Suitable for very large tables where the data has some natural ordering.
- Efficient for range queries and less storage-intensive.

□ **Expression Index:**

- An index on the result of an expression or function, rather than directly on column values.

□ **Partial Index:**

- An index that covers only a subset of rows in a table, based on a specified condition.

sql

```
CREATE INDEX idx_employee_last_name  
ON employees (last_name);
```

35. Cursor

a cursor is a database object used to retrieve a set of rows generated by a query and to process them one at a time. Cursors are particularly useful when dealing with large datasets where you want to process each row individually without loading the entire result set into memory at once.

Key Characteristics of Cursors

1. Row-by-Row Processing:

- Cursors allow you to fetch and process rows one at a time, which is useful for operations that require iterative processing of each row in a result set.

2. Memory Efficiency:

- By not loading the entire result set into memory, cursors help manage memory usage efficiently, especially with large datasets.

3. State Management:

- Cursors maintain their position within the result set, allowing you to fetch subsequent rows sequentially.

Declaring and Using Cursors

Cursors are typically used within PostgreSQL functions and stored procedures. Here's a basic outline of how to work with cursors:

1. Declare a Cursor:

- Define a cursor to hold the result set of a query.

2. Open the Cursor:

- Execute the query and establish the result set for the cursor.

3. Fetch from the Cursor:

- Retrieve rows from the cursor one at a time or in blocks.

4. Close the Cursor:

- Release the cursor and associated resources.

Example

Here's a detailed example demonstrating the use of a cursor in a PostgreSQL function:

1. Creating a Sample Table:

```
sql

CREATE TABLE employees (
    employee_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    hire_date DATE NOT NULL,
    department_id INTEGER,
    active BOOLEAN NOT NULL DEFAULT TRUE
);
```

2. Inserting Sample Data:

```
sql Copy code

INSERT INTO employees (first_name, last_name, email, hire_date, department_id, active)
VALUES
('John', 'Doe', 'john.doe@example.com', '2022-01-10', 1, TRUE),
('Jane', 'Smith', 'jane.smith@example.com', '2023-03-15', 2, TRUE),
('Alice', 'Johnson', 'alice.johnson@example.com', '2024-05-20', 3, FALSE);
```

3. Creating a Function with a Cursor:

```
sql Copy code

CREATE OR REPLACE FUNCTION process_active_employees()
RETURNS VOID AS $$
DECLARE
    emp_cursor CURSOR FOR
    SELECT employee_id, first_name, last_name
    FROM employees
    WHERE active = TRUE;
    emp_record RECORD;
BEGIN
    -- Open the cursor
    OPEN emp_cursor;

    -- Loop through each row
    LOOP
        -- Fetch the next row from the cursor
        FETCH emp_cursor INTO emp_record;

        -- Exit the loop when no more rows to fetch
        EXIT WHEN NOT FOUND;

        -- Process the row (for demonstration, we just raise a notice)
        RAISE NOTICE 'Processing Employee: % %', emp_record.first_name, emp_record.last_name;
    END LOOP;

    -- Close the cursor
    CLOSE emp_cursor;
END;
```

4. Calling the Function:

```
sql

SELECT process_active_employees();
```

36. SQLSTATE

Class P0 — PL/pgSQL Error	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
P0004	assert_failure

37. Performance Tuning and Optimization

MySQL performance tuning involves optimizing various aspects of your database system to **improve query execution speed, resource utilization, and overall responsiveness**. Here's a breakdown of key areas and techniques:

1. Query Optimization:

- **EXPLAIN statement:** Use EXPLAIN SELECT your_query to understand the query execution plan. It **reveals how MySQL intends to process the query, including which indexes it will use (or not use), the join order, and other crucial details**. Look for:
 - type: Ideally, you want ref, eq_ref, const, or system. Avoid ALL (full table scan).
 - possible_keys: Indexes that *could* be used.
 - key: The index actually chosen by MySQL.

- rows: The estimated number of rows examined. Lower is better.

1. Indexing Strategies and Types

1. Index Strategies

1. Choose Indexing Columns Wisely

- Prioritize columns used in:
 - WHERE, JOIN, GROUP BY, ORDER BY, and filtering clauses.
-

2 Composite Index vs. Single-Column Index

- **Composite Index:**
 - Use when queries filter or sort by multiple columns.
 - Example: (column1, column2).
 - **Single-Column Index:**
 - Use when queries frequently filter by a single column.
-

3 Avoid Over-Indexing

- Too many indexes slow down write operations (INSERT, UPDATE, DELETE).
 - Regularly review and remove unused indexes.
-

4 Use Partial Indexes

- Create indexes only for rows frequently queried.

Example:

```
CREATE INDEX idx_active_users ON users(status) WHERE status = 'active';
```

5 Covering Indexes for Performance

- Include all columns needed for a query to avoid accessing the table data.

Example:

```
CREATE INDEX idx_sales ON sales(customer_id, order_date, total_amount);
```

6 Index for Sorting and Filtering

- Index columns used in ORDER BY and WHERE clauses.

Example:

```
CREATE INDEX idx_order_date ON orders(order_date);
```

7. Avoid Indexing Low-Cardinality Columns

- Columns with few distinct values (e.g., gender) may not benefit from indexes.
-

8. Monitor and Adjust Indexes

- Use tools like **EXPLAIN (MySQL)** or EXPLAIN ANALYZE (PostgreSQL) to identify index usage.
 - Remove or reconfigure indexes based on query patterns.
-

2. Maintenance of Indexes

1. Rebuilding Indexes

- Periodically rebuild indexes to maintain performance.

Example in SQL Server:

```
ALTER INDEX idx_name ON table_name REBUILD;
```

2. Update Statistics

- Update index statistics to ensure the query planner has accurate data.

Example:

```
ANALYZE TABLE table_name;
```

3. Monitor Fragmentation

- Check and defragment indexes for better performance.
-

3. Tools for Index Optimization

- **MySQL:** EXPLAIN, SHOW INDEX.
 - **PostgreSQL:** EXPLAIN ANALYZE, pg_stat_user_indexes.
 - **SQL Server:** Query Execution Plans, Database Engine Tuning Advisor.
 - Third-party tools: SolarWinds, SQL Monitor.
-

4. Best Practices

1. **Index Frequently Used Columns:**
 - Use indexes on primary and foreign keys.
2. **Test Query Plans:**
 - Analyze performance with and without indexes.
3. **Monitor Query Patterns:**
 - Adjust indexes based on evolving queries.
4. **Combine Queries:**
 - Optimize queries to make better use of existing indexes.

2. Query Performance Analysis

Query performance analysis is the process of examining and improving the efficiency of database queries to reduce execution

time, optimize resource usage, and ensure scalability. It involves understanding the execution flow, identifying bottlenecks, and applying tuning techniques.

38. 1. Key Steps in Query Performance Analysis

3. 1.1 Identify Slow Queries

- Use database logs or monitoring tools to locate queries with high execution times or resource usage:
 - **MySQL:** Use the **Slow Query Log**.
 - **PostgreSQL:** Enable `log_min_duration_statement` to log slow queries.
-

4. 1.2 Analyze Execution Plans

Execution plans show how the database processes a query, including join strategies, index usage, and filter application.

- **MySQL:** Use the `EXPLAIN` or `EXPLAIN ANALYZE` command:

sql

Copy code

```
EXPLAIN SELECT * FROM employees WHERE department_id = 10;
```

- **PostgreSQL:** Use `EXPLAIN ANALYZE` for detailed execution times:

sql

Copy code

```
EXPLAIN ANALYZE SELECT * FROM employees WHERE department_id = 10;
```

- **SQL Server:** View execution plans in SQL Server Management Studio (SSMS).

Key Metrics to Look For:

- **Index Usage:** Check if indexes are being used effectively.
- **Joins:** Analyze join types (e.g., Nested Loop, Hash Join).

- **Full Table Scans:** Avoid scanning the entire table when possible.
 - **Cost:** Relative cost of each operation.
-

5. 1.3 Check Query Execution Time

Measure the time taken for a query to execute.

- MySQL:

sql

Copy code

```
SET PROFILING = 1;
```

```
SELECT * FROM employees WHERE department_id = 10;
```

```
SHOW PROFILES;
```

- PostgreSQL:

sql

Copy code

```
\timing
```

```
SELECT * FROM employees WHERE department_id = 10;
```

6. 1.4 Monitor Resource Usage

- Monitor CPU, memory, and disk I/O usage during query execution.
 - Use database-specific tools:
 - MySQL: Performance Schema.
 - PostgreSQL: pg_stat_statements.
 - SQL Server: Query Store.
-

39. 2. Common Query Performance Issues

7. 2.1 Missing Indexes

- Queries performing full table scans.

- Solution: Add indexes to columns in WHERE, JOIN, and ORDER BY clauses.
-

8. 2.2 Over-Indexing

- Too many indexes can slow down INSERT, UPDATE, and DELETE operations.
 - Solution: Regularly review and drop unused indexes.
-

9. 2.3 Poorly Written Queries

- Inefficient query structure or logic.
 - Examples:
 - Using SELECT * instead of selecting specific columns.
 - Applying functions on indexed columns (e.g., LOWER(column)).
 - Using subqueries instead of joins or Common Table Expressions (CTEs).
-

10. 2.4 Join Issues

- Inefficient joins, especially on large datasets.
 - Solution: Optimize join order and ensure appropriate indexes.
-

11. 2.5 High Cardinality Issues

- Filtering columns with many distinct values can strain indexes.
 - Solution: Consider composite indexes or partitioning.
-

12. 2.6 Locking and Blocking

- Long-running queries can cause locking, delaying other transactions.
- Solution: Reduce transaction scope and ensure proper indexing.

40. 3. Query Optimization Techniques

13. 3.1 Use Indexes Effectively

- Ensure that indexes cover columns in WHERE, JOIN, and ORDER BY.
-

14. 3.2 Optimize Joins

- Use the appropriate join type (e.g., INNER JOIN, LEFT JOIN).
- Ensure indexes on join columns.

Example:

sql

Copy code

```
SELECT e.employee_id, d.department_name
FROM employees e
INNER JOIN departments d ON e.department_id = d.department_id;
```

15. 3.3 Avoid Functions on Indexed Columns

- Rewrite queries to avoid applying functions to indexed columns.

Inefficient:

sql

Copy code

```
SELECT * FROM employees WHERE LOWER(first_name) = 'john';
```

Optimized:

sql

Copy code

```
SELECT * FROM employees WHERE first_name = 'John';
```

16. 3.4 Use LIMIT and Pagination

- Reduce the data fetched in queries with LIMIT.

Example:

sql

Copy code

```
SELECT * FROM employees ORDER BY employee_id LIMIT 100;
```

17. 3.5 Reduce Data Movement

- Minimize the number of rows and columns processed.

Example:

sql

Copy code

```
SELECT department_id, AVG(salary) FROM employees GROUP BY department_id;
```

18. 3.6 Partition Large Tables

- Partition tables to improve performance for queries targeting specific ranges.

Example (MySQL):

sql

Copy code

```
CREATE TABLE employees (  
    employee_id INT,  
    department_id INT,  
    ...  
) PARTITION BY RANGE (department_id) (  
    PARTITION p1 VALUES LESS THAN (100),  
    PARTITION p2 VALUES LESS THAN (200)  
);
```

19. 3.7 Use Query Hints

- Provide hints to the query optimizer for better execution.

Example (MySQL):

sql

Copy code

```
SELECT /*+ INDEX(idx_employee_id) */ * FROM employees WHERE  
employee_id = 101;
```

20. 3.8 Optimize Aggregations

- Use pre-aggregated data or materialized views for frequent reports.
-

41. 4. Tools for Query Performance Analysis

21. 4.1 Built-In Tools

- **MySQL:**
 - EXPLAIN, ANALYZE, and Performance Schema.
- **PostgreSQL:**
 - EXPLAIN ANALYZE and pg_stat_statements.
- **SQL Server:**
 - Query Store and Execution Plans.

22. 4.2 Third-Party Tools

- **SolarWinds Database Performance Analyzer:** Cross-platform database monitoring.
 - **New Relic:** Application performance monitoring.
 - **Datadog:** SQL query performance analysis.
-

42. 5. Monitoring and Benchmarking

- Regularly monitor query performance and system resource usage.

- Benchmark changes in a staging environment to validate improvements.

23. Execution Plans and Optimizer Hints

24. Execution Plans and Optimizer Hints

Execution plans and optimizer hints are essential tools for understanding and improving the performance of SQL queries. They provide insights into how a query is executed and allow developers to influence the query optimizer's decisions.

43. 1. Execution Plans

25. 1.1 What is an Execution Plan?

An execution plan outlines **how a database engine executes a query, detailing the steps, resources, and strategies it employs**. This helps identify bottlenecks and optimize query performance.

26. 1.2 Components of an Execution Plan

1. **Operations:** Actions like table scans, index lookups, joins, and aggregations.
 2. **Order of Execution:** Sequence of operations (e.g., filtering, sorting).
 3. **Costs:** Estimated resources (CPU, I/O) for each operation.
 4. **Rows:** Estimated or actual number of rows processed at each step.
 5. **Indexes:** Whether indexes are used and their effectiveness.
-

27. 1.3 Types of Execution Plans

1. **Estimated Execution Plan:**
 - Predicts how the query will run without actually executing it.
 - Useful for planning and understanding query behavior.
 - **Command:** EXPLAIN (MySQL, PostgreSQL).

2. Actual Execution Plan:

- Includes real execution statistics after running the query.
 - **Command:** EXPLAIN ANALYZE (PostgreSQL), Execution Plan in SQL Server Management Studio (SSMS).
-

28. 1.4 Generating Execution Plans

8. MySQL

- Use EXPLAIN to view the execution plan.

```
EXPLAIN SELECT * FROM employees WHERE department_id = 10;
```

- Use EXPLAIN ANALYZE (MySQL 8.0+) for actual execution statistics.

```
EXPLAIN ANALYZE SELECT * FROM employees WHERE department_id = 10;
```

9. PostgreSQL

- View an estimated execution plan:

```
EXPLAIN SELECT * FROM employees WHERE department_id = 10;
```

- View an actual execution plan:

```
EXPLAIN ANALYZE SELECT * FROM employees WHERE department_id = 10;
```

10. SQL Server

- Generate execution plans using SSMS:
 - **Estimated Plan:** Query → Display Estimated Execution Plan.
 - **Actual Plan:** Query → Include Actual Execution Plan → Execute Query.
-

29. 1.5 Common Patterns in Execution Plans

1. Full Table Scan:

- Indicates no index usage; consider indexing the relevant column(s).

2. **Index Scan:**

- Reads through the entire index; suitable for range queries.

3. **Index Seek:**

- Efficiently retrieves rows using an index; optimal for direct lookups.

4. **Nested Loop Join:**

- Suitable for small datasets but inefficient for large tables without indexes.

5. **Hash Join:**

- Efficient for large datasets; uses hashing for joining.
-

44. 2. **Optimizer Hints**

30. 2.1 **What are Optimizer Hints?**

Optimizer hints provide explicit guidance to the database query optimizer to influence its execution plan. These hints override the optimizer's default behavior.

31. 2.2 **Common Optimizer Hints**

11. **MySQL**

- **USE INDEX:** Forces the query to use specific indexes.

```
SELECT * FROM employees USE INDEX (idx_department) WHERE  
department_id = 10;
```

- **IGNORE INDEX:** Excludes specific indexes.

```
SELECT * FROM employees IGNORE INDEX (idx_department) WHERE  
department_id = 10;
```

- **FORCE INDEX:** Forces the query to use a specified index.

```
SELECT * FROM employees FORCE INDEX (idx_department) WHERE  
department_id = 10;
```

12. PostgreSQL

- **Enable/Disable Join Methods:** Adjust query behavior using session variables.

SET enable_hashjoin = OFF;

- **Parallel Query Hints:** Adjust parallelism for a query.

SET max_parallel_workers_per_gather = 2;

13. SQL Server

- **Query Hints:** Influence query behavior.
 - Force join type:
 - SELECT * FROM employees
 - INNER HASH JOIN departments
 - ON employees.department_id = departments.department_id
 - OPTION (HASH JOIN);
 - Optimize for a specific value:
 - SELECT * FROM employees
 - WHERE department_id = @dept_id
 - OPTION (OPTIMIZE FOR (@dept_id = 10));
 - **Table Hints:**
 - SELECT * FROM employees WITH (INDEX(idx_department));
-

32. 2.3 When to Use Optimizer Hints

1. **Query Optimizer Limitation:** When the optimizer consistently makes suboptimal choices.
 2. **Complex Queries:** Improve performance for edge cases or specific workloads.
 3. **Legacy Systems:** To work around limitations without altering database schema.
-

45. 3. Best Practices for Execution Plans and Hints

1. Understand Query Behavior:

- Always analyze execution plans before applying hints.

2. Use Hints Sparingly:

- Let the optimizer work as designed; hints should only address edge cases.

3. Monitor Query Performance:

- Use performance monitoring tools to evaluate the impact of hints.

4. Regularly Review Indexes:

- Ensure indexes align with query patterns to reduce the need for hints.

5. Test Changes:

- Validate execution plans in a staging environment before deploying to production.
-

46. 4. Tools for Execution Plan Analysis

1. MySQL:

- EXPLAIN, ANALYZE, Performance Schema.

2. PostgreSQL:

- EXPLAIN ANALYZE, pg_stat_statements.

3. SQL Server:

- Execution Plans in SSMS.

4. Third-Party Tools:

- SolarWinds, SQL Sentry, New Relic.
-

Let me know if you need deeper insights or examples on execution plans or optimizer hints!

47. Security and SQL Best Practices

1. Avoiding SQL Injection Vulnerabilities

Avoiding SQL Injection Vulnerabilities

SQL injection is a common and severe security vulnerability where attackers inject malicious SQL code into queries to manipulate or access sensitive data. Preventing SQL injection involves using secure coding practices and leveraging built-in protections provided by modern database systems.

1. Understanding SQL Injection

SQL injection occurs when user input is embedded directly into SQL statements without proper validation or sanitization.

2. Example of Vulnerable Code

```
String query = "SELECT * FROM users WHERE username = '" +  
username + "' AND password = '" + password + "'";
```

- If username is set to ' OR '1'='1 and password is empty, the query becomes:
- `SELECT * FROM users WHERE username = ' OR '1'='1' AND password = ''`;

This always evaluates to true, granting unauthorized access.

48. 2. Best Practices to Prevent SQL Injection

3. 2.1 Use Parameterized Queries (Prepared Statements)

Parameterized queries separate SQL code from data, ensuring user input is treated as a value rather than executable code.

Example (Java with JDBC):

```
String query = "SELECT * FROM users WHERE username = ? AND  
password = ?";
```

```
PreparedStatement stmt = connection.prepareStatement(query);
```

```
stmt.setString(1, username);
```

```
stmt.setString(2, password);
```

```
ResultSet rs = stmt.executeQuery();
```

Example (Python with SQLite):

```
query = "SELECT * FROM users WHERE username = ? AND password = ?"
```

```
cursor.execute(query, (username, password))
```

4. 2.2 Input Validation and Escaping

- Validate user input against expected formats (e.g., length, type, pattern).
- Escape special characters to neutralize potentially harmful input.

Example (PHP with MySQLi):

```
$username = $mysqli->real_escape_string($_POST['username']);
```

```
$password = $mysqli->real_escape_string($_POST['password']);
```

5. 2.3 Use Stored Procedures

Stored procedures are precompiled and limit dynamic SQL construction.

Example (MySQL):

```
DELIMITER //
```

```
CREATE PROCEDURE GetUser(IN username VARCHAR(50), IN password VARCHAR(50))
```

```
BEGIN
```

```
    SELECT * FROM users WHERE username = username AND password = password;
```

```
END //
```

```
DELIMITER ;
```

Usage:

```
CALL GetUser('john_doe', 'password123');
```

6. 2.4 Employ ORM Frameworks

Object-Relational Mapping (ORM) frameworks like Hibernate (Java), Entity Framework (.NET), and SQLAlchemy (Python) abstract query construction and use parameterized queries by default.

Example (SQLAlchemy in Python):

```
user = session.query(User).filter_by(username='john_doe',  
password='password123').first()
```

7. 2.5 Use Least Privilege Principle

- Restrict database user permissions to the minimum necessary.
 - Avoid using admin or superuser accounts for application-level queries.
-

8. 2.6 Limit Error Messages

- Avoid exposing database errors to users, as they may reveal database structure. **Example:** Instead of showing:

SQL Error: Syntax error near 'DROP TABLE'

Log the error internally and display a generic message:

An error occurred. Please try again later.

9. 2.7 Use Web Application Firewalls (WAF)

- Deploy a WAF to detect and block SQL injection attempts in web applications.
-

49. 3. Common Mistakes to Avoid

1. **String Concatenation:** Avoid constructing SQL queries with user input directly.
 2. **Relying on Escaping Alone:** Escaping is a mitigation, not a complete solution.
 3. **Ignoring Input Validation:** Always validate input even if using prepared statements.
-

50. 4. Advanced Techniques

10. 4.1 Use Query Parameterization in Dynamic SQL

If dynamic SQL is necessary, use parameterized queries even within dynamic query construction.

Example:

```
SET @query = CONCAT('SELECT * FROM ', table_name, ' WHERE id = ?');
```

```
PREPARE stmt FROM @query;
```

```
EXECUTE stmt USING @id;
```

11. 4.2 Implement Content Security Policies (CSP)

Protect against injection in client-side scripts that interact with databases indirectly.

12. 4.3 Database-Specific Security Features

- **MySQL:** Enable NO_BACKSLASH_ESCAPES to prevent escape sequences.
 - **PostgreSQL:** Use pgcrypto for encrypted storage and secure inputs.
-

51. 5. Tools for Detecting SQL Injection

1. **SQLMap:** Automated testing tool for detecting vulnerabilities.
 2. **Burp Suite:** Comprehensive web security testing tool.
 3. **OWASP ZAP:** Open-source penetration testing tool.
-

52. 6. Checklist for SQL Injection Prevention

- ☐ Use prepared statements and parameterized queries.
- ☐ Validate and sanitize all user inputs.
- ☐ Restrict database privileges.
- ☐ Avoid exposing detailed error messages.
- ☐ Employ security tools like WAFs.
- ☐ Regularly audit and test for vulnerabilities.

Let me know if you'd like examples in specific programming languages or more details on testing methods!

2. Secure SQL Coding Practices

3. Secure SQL Coding Practices

Writing secure SQL code is critical to protect databases from threats like SQL injection, data breaches, and unauthorized access. Adhering to secure SQL coding practices ensures robust data integrity and confidentiality.

53. 1. Input Validation and Sanitization

4. 1.1 Validate User Input

- Always validate inputs against expected formats and constraints.
- Examples:
 - Length restrictions: Ensure strings don't exceed the allowed size.
 - Type restrictions: Use data types like integers, dates, or enumerations.

Example (Validation in Python):

```
if not username.isalnum() or len(username) > 20:  
    raise ValueError("Invalid username")
```

5. 1.2 Sanitize Input

- Remove or encode potentially harmful characters (' , ; , --, etc.).

Example (PHP):

```
$username = htmlspecialchars($_POST['username'], ENT_QUOTES,  
'UTF-8');
```

54. 2. Use Parameterized Queries and Prepared Statements

Parameterized queries treat input data as values, not executable code.

Example (Java with JDBC):

```
String query = "SELECT * FROM users WHERE username = ? AND password = ?";
```

```
PreparedStatement stmt = connection.prepareStatement(query);
```

```
stmt.setString(1, username);
```

```
stmt.setString(2, password);
```

```
ResultSet rs = stmt.executeQuery();
```

Example (Python with SQLite):

```
query = "SELECT * FROM users WHERE username = ? AND password = ?"
```

```
cursor.execute(query, (username, password))
```

55. 3. Use Stored Procedures

Stored procedures predefine SQL code in the database, reducing the risk of injection.

Example (MySQL Stored Procedure):

```
DELIMITER //
```

```
CREATE PROCEDURE GetUser(IN username VARCHAR(50), IN password VARCHAR(50))
```

```
BEGIN
```

```
    SELECT * FROM users WHERE username = username AND password = password;
```

```
END //
```

```
DELIMITER ;
```

Usage:

```
CALL GetUser('john_doe', 'password123');
```

56. 4. Principle of Least Privilege

6. 4.1 Restrict Database User Privileges

- Assign the minimum required privileges to database users.
- Avoid using root or admin accounts for application-level queries.

Example (Granting Minimal Privileges):

```
GRANT SELECT, INSERT, UPDATE ON employees TO  
'app_user'@'localhost';
```

7. 4.2 Use Role-Based Access Control (RBAC)

- Assign roles with predefined permissions to database users.

Example (PostgreSQL):

```
CREATE ROLE read_only;  
GRANT CONNECT ON DATABASE mydb TO read_only;  
GRANT SELECT ON ALL TABLES IN SCHEMA public TO read_only;
```

57. 5. Avoid Dynamic SQL

Dynamic SQL, when improperly handled, increases the risk of SQL injection. If unavoidable, use parameterized queries.

Avoid:

```
String query = "SELECT * FROM users WHERE username = '" +  
username + "'";
```

Use:

```
String query = "SELECT * FROM users WHERE username = ?";  
PreparedStatement stmt = connection.prepareStatement(query);  
stmt.setString(1, username);
```

58. 6. Limit Data Exposure

8. **6.1 Use SELECT Instead of SELECT **

- Fetch only the necessary columns to reduce exposure.

Example:

```
SELECT username, email FROM users WHERE user_id = 10;
```

9. 6.2 Mask Sensitive Data

- Encrypt sensitive data using database-specific features (e.g., AES encryption).

Example (MySQL AES_ENCRYPT):

```
INSERT INTO users (username, email, password) VALUES ('john_doe',  
'john@example.com', AES_ENCRYPT('password123',  
'encryption_key'));
```

59. 7. Protect Against SQL Injection

10. 7.1 Use Escape Functions

- Escape special characters to prevent code injection.

Example (MySQLi in PHP):

```
$sanitized_username = $mysqli->real_escape_string($_POST['username']);
```

11. 7.2 Disable Dangerous Features

- Avoid using database features like xp_cmdshell in SQL Server unless absolutely necessary.
-

60. 8. Secure Error Handling

- Hide detailed error messages to avoid exposing database structure.

Example:

try:

```
cursor.execute(query)
```

except Exception as e:

```
log_error(e) # Log internally
```

```
return "An unexpected error occurred."
```

61. 9. Regular Auditing and Monitoring

12. 9.1 Log SQL Queries

- Log query execution for auditing purposes.

Example (PostgreSQL):

```
log_statement = 'all';
```

13. 9.2 Use Database Monitoring Tools

- Tools like SolarWinds or pg_stat_statements help track query performance and detect anomalies.
-

62. 10. Additional Best Practices

14. 10.1 Limit Query Results

- Use LIMIT or pagination to avoid returning excessive data.

Example:

```
SELECT * FROM employees LIMIT 100;
```

15. 10.2 Avoid Hardcoded Credentials

- Store database credentials in environment variables or secure vaults.

Example (Python):

```
import os  
db_user = os.getenv("DB_USER")  
db_pass = os.getenv("DB_PASS")
```

16. 10.3 Patch and Update Regularly

- Keep database software and libraries updated to address known vulnerabilities.
-

63. 11. Tools for Secure SQL Development

1. **OWASP ZAP**: Detect SQL injection vulnerabilities.
 2. **SQLMap**: Automated SQL injection testing.
 3. **Static Code Analysis Tools**: Detect insecure SQL coding practices.
-

By adhering to these secure SQL coding practices, you can significantly reduce the risk of vulnerabilities and safeguard your database and sensitive data. Let me know if you want examples for specific scenarios or languages!

3.Database Encryption and Data Masking

4. Database Encryption and Data Masking

Database encryption and data masking are two critical techniques for protecting sensitive data. They ensure data confidentiality and privacy, even if unauthorized access occurs. Below, we explore their concepts, implementation strategies, and best practices.

64. 1. Database Encryption

5. 1.1 What is Database Encryption?

Database encryption involves converting plaintext data into ciphertext using encryption algorithms. Only authorized parties with decryption keys can access the original data.

6. 1.2 Types of Database Encryption

14. 1.2.1 Transparent Data Encryption (TDE)

- Encrypts data at rest (data stored on disk).
- Automatically encrypts/decrypts data as it is read from or written to storage.
- Example: SQL Server TDE, Oracle TDE.

SQL Server TDE Example:

```
CREATE MASTER KEY ENCRYPTION BY PASSWORD =  
'StrongPassword';  
  
CREATE CERTIFICATE MyServerCert WITH SUBJECT = 'MyCert';  
  
CREATE DATABASE ENCRYPTION KEY  
    WITH ALGORITHM = AES_256  
    ENCRYPTION BY SERVER CERTIFICATE MyServerCert;  
  
ALTER DATABASE MyDatabase SET ENCRYPTION ON;
```

15. 1.2.2 Column-Level Encryption

- Encrypts specific columns containing sensitive data.
- Offers more granular control than TDE.

MySQL Example (AES Encryption):

```
INSERT INTO users (username, ssn) VALUES ('JohnDoe',  
AES_ENCRYPT('123-45-6789', 'encryption_key'));
```

Decryption:

```
SELECT AES_DECRYPT(ssn, 'encryption_key') FROM users WHERE  
username = 'JohnDoe';
```

16. 1.2.3 File-Level Encryption

- Encrypts entire database files using tools like BitLocker or Linux LUKS.

17. 1.2.4 Application-Level Encryption

- Encrypts data at the application layer before storing it in the database.
 - Ensures data is encrypted during transmission and storage.
-

7. 1.3 Best Practices for Database Encryption

1. Use Strong Encryption Algorithms:

- AES-256, RSA, or elliptic curve cryptography.

2. Secure Key Management:

- Store encryption keys separately from encrypted data.
- Use hardware security modules (HSMs) or secure vaults (e.g., AWS KMS).

3. **Encrypt Backups:**

- Ensure database backups are encrypted to prevent data leakage.

4. **Encrypt in Transit:**

- Use SSL/TLS for communication between applications and the database.
-

65. **2. Data Masking**

8. 2.1 What is Data Masking?

Data masking replaces sensitive data with fictitious or scrambled data, making it unreadable while retaining its usability for testing or development purposes.

9. 2.2 Types of Data Masking

18. 2.2.1 Static Data Masking

- Permanently masks data in non-production environments.
- Example: Scrambling names or SSNs in a test database.

19. 2.2.2 Dynamic Data Masking (DDM)

- Masks data dynamically during query execution without altering the actual data stored in the database.
- Example: SQL Server DDM.

SQL Server Example (DDM):

```
ALTER TABLE customers
```

```
ALTER COLUMN email ADD MASKED WITH (FUNCTION = 'email()');
```

Query Output:

- **Admin:** Shows full email.
- **Non-admin:** Displays a***@domain.com.

20. 2.2.3 Tokenization

- Replaces sensitive data with unique tokens that map to the original data.

21. 2.2.4 On-the-Fly Masking

- Temporarily masks data during real-time data access, often used for reporting.
-

10. 2.3 Use Cases for Data Masking

1. Test/Development Environments:

- Prevents accidental exposure of sensitive data.

2. Compliance:

- Meets privacy regulations like GDPR, HIPAA, or PCI DSS.

3. Outsourcing:

- Safeguards sensitive data when shared with third-party developers.
-

11. 2.4 Best Practices for Data Masking

1. Identify Sensitive Data:

- Locate columns with PII (Personally Identifiable Information) or financial data.

2. Apply Role-Based Access Control (RBAC):

- Ensure only authorized roles can view unmasked data.

3. Use Consistent Masking Techniques:

- Maintain data integrity and usability for testing.

4. Automate Masking Processes:

- Use tools like Delphix, Oracle Data Masking, or Informatica.
-

66. 3. Database Encryption vs. Data Masking

Feature	Database Encryption	Data Masking
Purpose	Protects data confidentiality.	Obscures data for non-production use.
Where Applied	Storage (data at rest).	Query results or copied datasets.

Feature	Database Encryption	Data Masking
Use Case	Prevent unauthorized access to data.	Enable safe use of data in testing or analytics.
Impact on Performance	Minimal for TDE; may vary for column-level encryption.	Minimal for static masking; can add overhead for dynamic masking.
Compliance	HIPAA, GDPR, PCI DSS.	GDPR, PCI DSS, anonymization laws.

67. 4. Tools for Database Encryption and Data Masking

12. 4.1 Encryption Tools

- **Database Built-In Tools:**
 - SQL Server TDE, Oracle TDE, MySQL AES.
- **External Tools:**
 - AWS KMS, Azure Key Vault, Vault by HashiCorp.

13. 4.2 Data Masking Tools

- **Built-In Features:**
 - SQL Server Dynamic Data Masking, Oracle Data Redaction.
 - **Third-Party Tools:**
 - Delphix, Informatica Data Masking, Datprof.
-

68. 5. Implementation Strategy

1. Assess Data Sensitivity:

- Identify sensitive fields requiring encryption or masking.

2. Choose the Right Approach:

- Use encryption for confidentiality and masking for non-production use.

3. Implement Gradually:

- Start with less critical systems to minimize risk during deployment.

4. **Test Performance:**

- Evaluate query performance post-implementation.

5. **Monitor and Audit:**

- Regularly review encryption keys and masking policies.
-

Let me know if you need detailed implementation examples or help with specific databases!